THE UNIVERSITY
of EDINBURGH

Quantitative Bounds on the Security-Critical Resource Consumption of JavaScript Apps

Daniel Franzen

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2016

Abstract

Current resource policies for mobile phone apps are based on *permissions* that unconditionally grant or deny access to a resource like private data, sensors and services. In reality, the legitimacy of an access may be context-dependent - for example, depending on how often a resource is accessed and in which situation. This thesis presents research into providing bounds on the access of JavaScript apps to security and privacy-relevant resources on mobile devices. The investigated bounds are *quantitative* and *interaction-dependent*: for example, permitting one access each time the user presses a specified button.

Two novel systems are presented with different approaches to providing these bounds. The system *PhoneWrap* injects a quantitative policy into an app and enforces the bound dynamically during runtime by monitoring the resource consumption and the user interaction. If the injected bound is exceeded, the resource request is replaced by a deny action. This way, PhoneWrap restricts the unwanted behaviour while the expected functionality can be performed. Policies for this system describe the UI elements which trigger the expected resource consumption and the number of resource units consumed for each interaction. The enforcement of the policies is achieved via wrapping the critical APIs using JavaScript internal features. The injection of a policy can be performed automatically. PhoneWrap is the first system using the lightweight wrapping method to inject policies directly into mobile apps and the first to combine quantitative policies with interaction-dependencies.

The second system *AmorJiSe* statically analyses the resource consumption of a given JavaScript program. This system automatically infers amortised annotations on top of given JavaScript data types. The amortised annotations symbolise reserved resource units stored in the data structures. This way the amount of resource units available to the app is expressed dependent on the size of the data structures. The resulting function types of the UI handlers can be used to extract interaction-dependent bounds. The correctness of these bounds is proven in relation to a resource-aware operational semantics. AmorJiSe extends the known amortised type paradigm to JavaScript with its dynamic object structures and applies this paradigm to the novel domain of mobile resources.

Although, the two systems are based on similar resource models and produce similar resource bounds, they use different methods with different properties which are presented in this dissertation.

Lay summary

Apps on smartphones request a set of permissions during installation to access the private data (Contacts, accounts, emails), the sensors (location, camera, microphone) and services (phone calls, messages, playing music). During installation the user cannot know how these resources are used by the program. Abuse of these resources in the wrong context poses a threat to the users security or privacy

This dissertation aims to provide more information and control how the requested resources are used. Instead of the permission, which can only grant full access at any time, the systems presented here provide a bound on how often the resource is used and connect the use to a specific action of user for example a specific button.

This is achieved by two different methods. The system PhoneWrap alters the behaviour of an app in a way that the app cannot use the resource more often than specified in the bound. This is achieved by granting the app the correct number of one-time-tickets, which the app has to pay every time it accesses the resource. If the app tries to use the resource without paying a ticket the access is replaced by a suitable deny behaviour.

The other system AmorJiSe analyses the app before it is executed and provides information on how oft and connected to which user interactions the resource is used. The results of this analysis are mathematically proven for all possible behaviours of the app.

Although the two systems provide similar results, the methods have different properties and are therefore suited for different situations.

Acknowledgements

My first thanks is due to David Aspinall, for all the discussions, patience and the valuable advice - technical, formal and otherwise - I got from him. I am grateful for the early stage discussions with Sergio Maffeis and Gareth Smith which helped me see the big picture and with Martin Hofmann which helped to fill in the details of the type system.

I would also like to thank my many friends, especially in New Scotland for providing the right amount of distraction and making my time in Edinburgh an incredible one.

A special thanks to Melanie for all the support she has given me during the last few years, for tolerating my mood if it did not go well and for being the zeroth reader for a lot of my work.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Preliminary versions of the results presented in Chapter 4 and 5 have been published as [39] and [40].

(*Daniel Franzen*)

Table of Contents

Right, my phone. When these things first appeared, they were so cool. Only when it was too late did people realize they are as cool as electronic tags on remand prisoners.

David Mitchell

Chapter 1

Introduction

1.1 Motivation

Phones and other mobile devices are permanent companions in our modern lives. As such they are filled with personal information, secrets used for authentication or payments and their services, such as text messaging, generate additional fees. Smartphones are equipped with a variety of sensors, cameras and microphones which could record every detail of the user's life. Their huge potential make them a valuable target for advertisers and malicious software as recognised by security software vendors and security advisers:

> *30% of Android apps may expose users to data loss and privacy violations*
> (Marble Labs Mobile App Threat Report for January 2014)

> *[...] during the second half of 2015, only 18% of new and updated apps were benign, 30% were moderate or suspicious, and the remaining 52% were unwanted or malicious* (Webroot "2016 Threat Brief")

> *[...] recent polls show that 9 in 10 Americans feel they have in some way lost control of their personal information*
> (The White House, Office of the Press Secretary, Fact sheet Jan 2015)

Operating system for smartphones have recognised this threat and protect the critical resource to mitigate the potential damage of malicious apps. The prevalent such system is Android with around 80% market share. The resource access control in Android and similar systems is based on permissions. The app requests a set of permissions each of which, if granted by the user during installation, grants unrestricted access to a specific resource. However, there is no fine-grained control on how often and in which context the app may use a granted resource. A previous study [36] shows

that 93% of the free apps ask for at least one potentially dangerous permission. Users often have to guess for which functionality of the app a permission is requested and accept the risk of the permission being abused. Contrary to the perception that phone users do not care about the permissions, the following statements show the reaction when users discover resource access in a context or quantity they did not expect:

> *The app requests permission to access and record audio without my confirmation and to access my camera and take video or pictures without my confirmation. Why do they need this capability?*
>
> (Facebook Community forum)

> *You say "It's not that these are randomly turning on and taking pictures and/or recording convos" but the thing is by giving carte blanche approval they can.*
>
> (Facebook Community forum)

> *Why would whatsapp access my contexts over 7000 times when I only opened the app maybe less then 100 times? [...] Deleted that app is what I did.*
>
> (Review of "DTEK by BlackBerry" on Google Play)

> *For the life of it, I cannot see any rational reason why a ride-sharing app would require to pry into your browsing history, much less other apps that you use or are running.*
>
> (reddit /r/privacy)

These show that, for some users, more information and control over the resource access is needed. However, asking the user for every access request breaks the workflow and is not acceptable for the user.

The missing control can be provided in the form of interaction-dependent bounds on the resource access. Modern apps are interaction-centric: during launch they only initialise and display the elements of the user interface. Afterwards, every time the user interacts with one of those elements, the requested functionality is executed. Therefore, access to the correct resources in the appropriate amount for each functionality can be associated to the UI elements. Ideally, the app is not granted the permission to access a resource unconditionally, but may access the resource a bounded number of times every time the user activates a specific functionality.

Many security and privacy concerns for mobile systems can be considered as a *resource problem*: messaging, phone calls and other billable services are resources by themselves and bounds on their usage limit the bills the app generates; location tracking accesses the GPS sensors and the data connection and bounds on the access to these

resources ensure the private location information is only accessed when justified by the functionality; private data leakage can be limited to the amount the user is comfortable to provide to a specific app by considering the content (pictures, documents or contact data) stored on the phone as resource; denial-of-service attacks exceed the capacity of the attacked resource and bounds can successfully counteract them; buffer overflows over-use the resource memory space and a bound on the memory usage makes it possible to prevent or anticipate potential threats.

To protect the user in all these scenarios, security frameworks are required to provide more fine-grained bounds on the permitted access.

As programming language for mobile apps, JavaScript is becoming more and more important. JavaScript has been the dominant language for web applications and, since many mobile apps perform functionality similar to web applications, it is an obvious choice for mobile apps. Emerging mobile operating systems like Tizen [38] and ChromeOS [60] build directly on JavaScript as their primary app programming language. For Android and other non-JavaScript operating systems various frameworks, e.g. Adobe PhoneGap, Sencha Touch or ratchet[1], package JavaScript into native apps to install and execute HTML / JavaScript apps seamlessly. With over 400,000 developers and over 1 million apps, Adobe PhoneGap [59] is the most popular such framework. It executes the JavaScript code on all major mobile platforms in an instance of the built-in browser of the operating system. The same JavaScript app can be packaged for multiple different operating systems this way and apps can even re-use code from a web page.

On top of the standard JavaScript language features, PhoneGap offers plugins which access the native capabilities and sensitive resources of the mobile phone. These plugins are naturally governed by the default access policy of the operating system, but, as discussed above, permissions are not fine-grained enough to efficiently guard the resources. If a permission has been granted, the app and the contained third party code can use all known exploits for JavaScript to abuse the granted resources. The Microsoft Security Intelligence Report [86] identified JavaScript exploits as “the most commonly encountered type of exploits with an encounter rate more than four times as high as the next most common type of exploit.” For this reason methods to provide more control over the resource usage of such apps are even more important.

Since JavaScript is interpreted during runtime, JavaScript apps include the full source code of the app. This presents an opportunity to provide those bounds.

[1] see Section 2.1.4 for more details

1.2 Problem definition

This thesis considers the following research hypothesis:

> *By using static analysis or dynamic enforcement it is possible to automatically impose quantitative bounds on the resource consumption of mobile apps written in JavaScript which allow the resource access required for the wanted features while rejecting unreasonable resource usage.*

The hypothesis differentiates between two components of the app's behaviour. Its *wanted functionality* are the actions expected or requested by the user. On top of that, many apps provide *unwanted functionality* which could be intrusive ads, malware or additional features not required by the individual user. An indication whether a feature is wanted or unwanted are the user interactions with the app. By interacting with the app's user interface the user activates features and, therefore, implicitly confirms the legitimacy of the following actions.

Each wanted feature of an app *requires* different amounts of resources. Some functionalities require one special kind of resource (like sending an SMS) other require a larger set of different resources. A policy maps each functionality to its required amount for each resource. The specific values in the policy can depend on the user. Therefore, users need to be able to specify individual policies to capture their individual resource requirements.

Apart from picking or creating the correct policies, the restriction should be applied *automatically*. This includes, in particular, that the user does not have to understand or annotated the source code of the app or the restriction. Manipulating the source code exceeds the capabilities of the average mobile phone user and would therefore restrict the user base to a small set of experts.

Two principle ways of *imposing a bound* on an mobile app will be considered: an app can either be *checked* against a given policy before installation and rejected in case it does not comply or an app can be *prevented* from exceeding the bound during the execution and stopped or altered if the bound is exceeded.

To capture the various resource scenarios, the resource model in this work uses abstract resource units. The concrete instantiation of a resource unit depends on the resource and use case. For example, each file system access might be considered

one resource unit or, to be more fine-grained, the units might be proportional to the amount of accessed data. By using this abstraction, the proposed methods are capable of handling all security issues mentioned above and many more resource problems.

The specification of the bound has to be flexible enough to separate wanted from unwanted resource behaviour for real-world apps. The bounds may be *interaction-dependent* to reflect the interaction-centric design of modern apps. They describe which user actions trigger resource access and how many units may be consumed in response to each interaction. When the performed functionality depends on the user's input, the bounds have to be *data-dependent*, for example, depending on the size of an input.

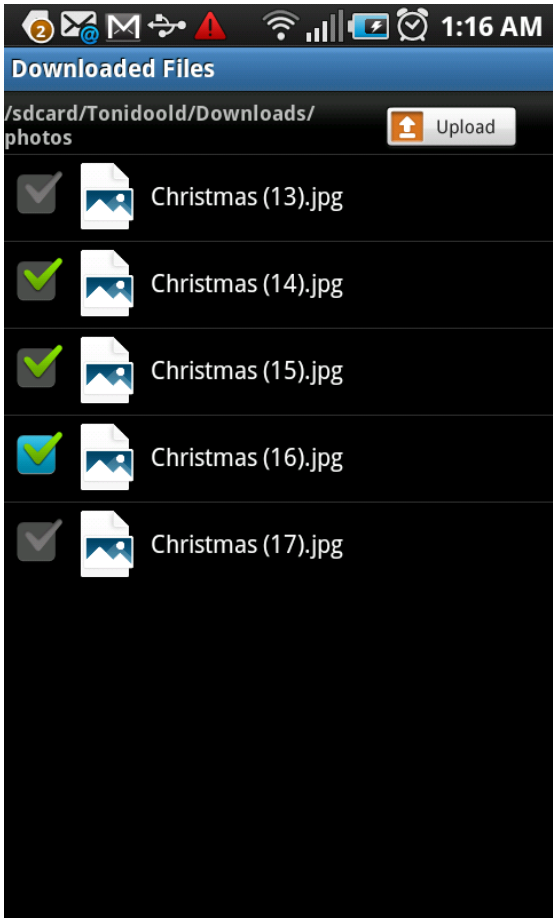
The quality of a given bound highly depends on the resource, the expected functionality of the app, on the device and the user itself. For example, if on the device each sent SMS is billed, even small numbers may be unacceptable, especially to premium rate numbers. On devices with an unlimited message allowance, larger bounds suffice. Some users might consider a scan through the contacts and a consequent upload to the cloud as breach of privacy, others consider occasional scans as acceptable functionality to find friends using the same app. Company issued devices might have a different demand on the resource restrictions than private devices. The policy author, which could, for example, be the end user, device administrator or a security provider, has to take this whole context into account to describe the *reasonable usage*.

1.3 Example

As a simple example, consider a fictional Android app *Pic-Up* which allows to upload pictures from the external storage of the phone onto a public sharing web page. The app scans the external storage for pictures and then presents a list with all found pictures to the user. The user can select one or more of the presented pictures and press "upload" to queue all selected pictures for upload. As soon as a data connection to the server becomes available, Pic-Up uploads the queued pictures. This functionality justifies access to the external storage, which is guarded by the Android permission `READ_EXTERNAL_STORAGE`.

The expected resource behaviour of the app includes one access to the external storage during initialisation to scan for uploadable pictures. Furthermore, once the upload capability becomes available, the app accesses each selected picture once. However, due to the granted permission `READ_EXTERNAL_STORAGE`, Pic-Up is able to read all

Figure 1.1 Example: Pic-Up



(Screenshot taken from Tonido app, as example how Pic-Up's UI could look like.)

files on the storage at any time including all downloaded files, the documents, the stored music and video files. Imagine a similar but malicious app *Pic-Down* which performs the same functionality but additionally uploads all pictures and private documents found on the device onto a malicious server. In the app-store both apps look equal, since they require the same permissions, but Pic-Down potentially leaks the user's confidential documents and pictures. This kind of attack is common with Android apps. A survey [80] found that 90% of the most popular apps have a fake app which adds malicious behaviour like data theft or premium service abuse.

The expected resource access of the app's advertised functionality can be described with interaction-dependent bounds. Pick-Up has to access the storage once at the beginning and once for each selected picture. Therefore, if the user in multiple sessions first marks c_1 pictures for upload, then c_2 ... and finally c_k pictures, the number of legitimate accesses can be bounded by $1 + c_1 + ... + c_k$. The malicious behaviour of Pic-Down exceeds this bound. The bounds can therefore help to differentiate the ex-

pected behaviour from the potentially malicious one.

1.4 Methods and goal

This thesis presents two approaches to bound the resource usage of JavaScript apps, the dynamic system PhoneWrap and the static system AmorJiSe. The two systems provide the bounds with different methods:

1.4.1 PhoneWrap

The dynamic system PhoneWrap inserts a resource monitor into a PhoneGap app package and mediates access to the guarded resource.

Given an app package, PhoneWrap inserts a script which instruments all critical resource APIs at the top of the main HTML file of the app. The instrumentation monitors the resource consumption and user interaction during runtime and executes an appropriate deny behaviour if the resource consumption exceeds the bound specified in the policy. The isolation of the enforcement script from the application code is achieved by JavaScript function scopes and, hence, enforced directly by the JavaScript engine. The modified app is packaged into an Android package and re-signed with a policy key[2]. As a result, the modified app can be executed on an unmodified mobile phone.

PhoneWrap can be used either by the end user itself, by the app developer or a third party and the policy files can be shared by experienced policy providers similar to the lists currently used for ad-blockers. This way, less experienced smartphone users can choose a policy which fits their usage scenario of the app and inject it directly into the downloaded app. PhoneWrap supports the policy author by extracting all necessary information from the app package and helps to select the UI elements which trigger the resource accessing functionality via normal interaction with the app. This way, the policy can be specified without deep knowledge about the implementation details of the app or PhoneWrap.

PhoneWrap can even be used to insert multiple different policies into an app. As a result, a resource access is only granted if each policy grants access individually. This is useful when different parties (e.g. app developer, app distributor, device administrator and device user) have different requirements on the resource behaviour or multiple resources have to be guarded.

[2]For a more detailed discussion see Section 4.4

1.4.2 AmorJiSe

The static system AmorJiSe uses type inference to *statically infer* bounds on the resource usage of given JavaScript source code. AmorJiSe's analysis can be performed a-priori, ahead of installing the app so that the user can avoid even partial execution of unwanted behaviours. It can also be used to compare the resource need of different apps with similar functionality.

AmorJiSe infers types for all subexpressions of the analysed code including information about their resource usage. The resulting types for the input values of a program imply data-dependent resource bounds and the function type of the input handlers imply interaction-dependent bounds. These bounds are mathematically proven correct in the sense that the provided resource-aware execution model does not consume more resources during the evaluation of an analysed expression than specified by the bound. Rather than defining yet another type system for JavaScript, the types of AmorJiSe are designed to extend an existing type system. The added resource inference layer of AmorJiSe itself is designed to be fully automatic and strict: the inference does not require additional annotations in the JavaScript source code and the results may conservatively over-estimate the behaviour of all possible executions. However, depending on the underlying type system, the combined system can benefit from additional information such as source-code annotations or allow unsound under-approximations under certain conditions.

AmorJiSe infers the types via reduction to a linear programming problem (LPP) which can be solved with conventional solvers. The result can also be used as a certificate or "digital evidence" [11, 107] for the analysis: either the developer or a third party, for example the distributor of the app, can execute the analysis ahead of time and supply a certificate consisting of the types and the solution for the LPP together with the app. During installation, the device can then check the code against the structure of the provided types and, while doing so, re-create the LPP on the annotations. The provided solution to the LPP can then be checked efficiently. Since neither the type inference nor the LPP solver have to be re-executed, the certificate verification is suitable even for devices with less computing power like smartphones.

1.5 Outline and contributions

The three main contributions presented in this thesis are:

1. PhoneWrap and its prototype implementation which is, to the best of my knowledge, the first system to combine quantitative and interaction-dependent resource policies and enforce these policies in JavaScript apps using a runtime wrapper.

2. A detailed investigation of JavaScript typing mechanisms, including a counter example pointing out a previously unknown flaw in the type checking algorithm of the type system JS_0^T [8] and a proposed fix.

3. AmorJiSe, the first amortised type system for a core of JavaScript. It shows how sound amortised types can be inferred in the presence of the dynamically changing object structures of JavaScript.

The remainder of this thesis is structured in the following way:

Chapter 2 introduces JavaScript and the JavaScript mobile app framework PhoneGap with the focus on the challenges for analysis. This chapter also provides an overview of resources accessible to PhoneGap apps and surveys the related work on resource analysis.

Chapter 3 presents the formal description of the novel interaction-dependent ticket-based policies as enforced by PhoneWrap including the policy description, policy state and transitions. Furthermore, it outlines the formal definition of wrapping as an enforcement method and proves the soundness of the formal model.

Chapter 4 presents the dynamic system PhoneWrap which enforces quantitative resource policies by wrapping the resource relevant APIs with a policy wrapper. It introduces the methods and the different steps of the policification process consisting of unpacking, analysis, policy creation, policy injection and repackaging of the apps for installation. To show the feasibility of this approach, the tool is applied to real-world apps downloaded from the Google Play Store.

Chapter 5 presents the discussion of existing type systems for JavaScript in preparation for the system AmorJiSe. Different properties of type systems and requirements for the underlying type system of AmorJiSe are discussed. Based on those properties the existing type systems are examined and common methods and differences extracted. The conclusion of this chapter is that the system JS_0^T [9] by Anderson et al. is the best foundation for AmorJiSe.

Chapter 6 introduces AmorJiSe, an amortised extension for a static type system for JavaScript. The chapter presents the basic type system, its properties and the type inference algorithm. Furthermore a formal resource annotated semantics is given against which AmorJiSe is evaluated. The system is parameterised by the resource model,

which makes it possible to reason about different resources. The main result of this section is the formal proof of soundness for AmorJiSe against the specified operational semantics.

Chapter 7 discusses the chosen underlying system JS_0^T and presents the flaw in its type checking soundness proof. This flaw is discussed and a full counter example provided which evaluates to a runtime error but is typed as value. Therefore, the type checking system cannot have the claimed soundness property. This chapter furthermore shows how an adaptation of the type inference rules can be used to fix the flaw in the type checking rules.

Finally, Chapter 8 closes the thesis with a conclusion, comparing the two systems, and final remarks about future work on this topic.

Chapter 2

Background

This chapter discusses the background for the research results presented in this thesis. In particular, it presents the language features of the JavaScript language and its challenges for analysis, the structure of JavaScript apps and the resources accessible by them. Furthermore, this chapter discusses previous research in the area of the JavaScript language, its use in the real world, related analysis efforts and mobile resource policies.

2.1 The JavaScript language

JavaScript was introduced in 1995 by Brendan Eich at Netscape (see [105]) as language for web applications in browsers. It was first standardised in 1997 as ECMAScript [61]. The language is currently published in version 6 (ES6), but versions 5 (ES5) and 3 (ES3) of the language are still widely used. The current standard consists of over 500 pages of prose. An official reference implementation for JavaScript does not exist. Instead, each browser implements the standard supporting the majority of the features, but differing in some details.

The language is imperative and object-oriented with inheritance based on prototypes instead of classes. It has various built-in libraries, e.g. to interact with the HTML DOM tree. JavaScript was developed as a scripting language, with the aim to implement client-side features for web pages quickly. For this reason, JavaScript executes many expressions, which would result in runtime errors and termination in other languages. However, the behaviour of such expressions is often surprising for inexperienced JavaScript developers. With version 5, JavaScript introduced the *strict mode*(ES5S) to improve the semantics of JavaScript without breaking the backwards

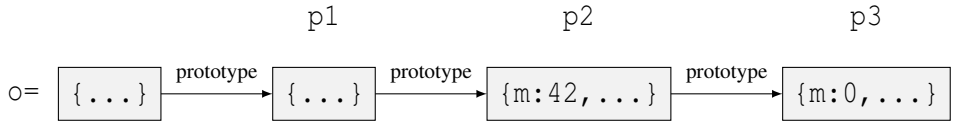
compatibility of older programs. It can be optionally enabled by including the string "strict mode" as the first expression in a JavaScript file or function body and disables the often criticised features causing the scope to be non-static: in strict mode the `with` operator is no longer available, `eval` cannot declare new variables and a few other features allowing the program to access the scope directly, like the binding of `this` to the global scope object in functions, are more restricted.

The following section presents a summary of JavaScript's features which pose a challenge for analysis of JavaScript apps. They illustrate why analysis techniques for other language cannot be applied directly and, instead, JavaScript analysis is a research area by itself. The features discussed here apply to version 3 and the normal mode of versions 5 or later. Differences in the strict mode are mentioned separately.

2.1.1 Language features

2.1.1.1 Prototypal inheritance

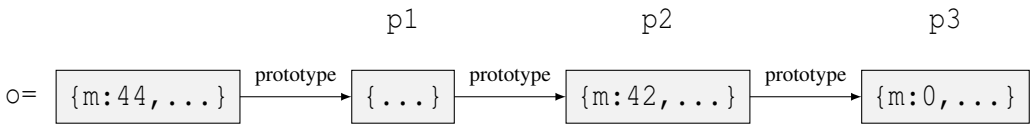
The feature that separates JavaScript distinctively from other object-oriented programming languages, such as C or Java, is the inheritance mechanism which JavaScript handles via prototype chains. Objects are not created from classes which inherit from each other, but each object instance inherits from another object instance, called its *prototype*. This prototype can itself have a prototype which results in a prototype chain. During field lookup, the JavaScript runtime engine scans through this chain to find the requested field. Assume an expression tries to access the field `m` of an object `o`:



First, the object `o` itself is scanned for the field `m` and, if found, the stored value is returned. If the field is not present in the object itself, the search proceeds at the prototype, until the field is either found or an object without a prototype is reached in the prototype chain. In the latter case the value `undefined` is returned.

In the example here, assuming the objects `o` and `p1` do not contain the field `m`, the field `m` is found at the second prototype `p2`, returning the value `o.m=42`. The JavaScript standard only allows to set the prototype link via the `prototype` property of the constructor of an object. However, in all major JavaScript implementations the prototype can be manipulated directly using the `__proto__` property of an object.

The prototypes are not involved in assignment. In an expression like `o.m=44` the direct field `m` of the object `o` is either overwritten or created, independent of the field `m` in any of the prototypes. Thus, in this example, the field `m` is created in the object `o`, resulting in the following prototype chain:

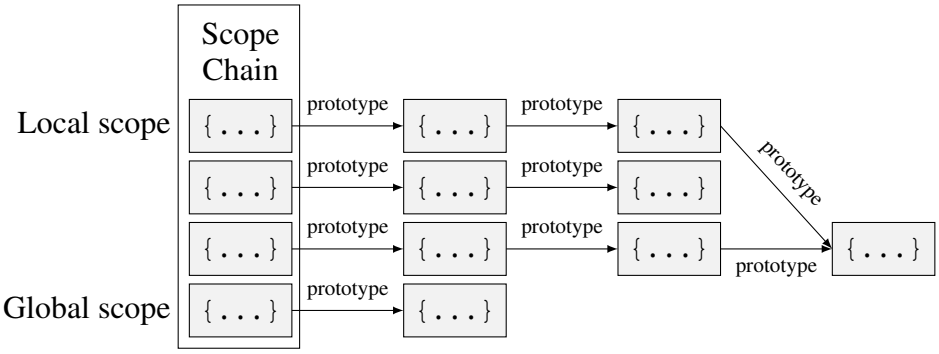


Afterwards the field `m` of the object `p2` is shadowed by `o.m` and not accessible directly through `o` anymore. Only when the field `m` of the object `o` is deleted later, the field `m` of `p2` becomes visible for `o` again.

2.1.1.2 Non-static scope

The values (or references to the values in the heap) of variables in JavaScript are stored in a *scope chain* consisting of scope objects. This chain begins with the global scope and ends in the most recent scope object which contains the local variables. Each of the scope objects is handled as a standard JavaScript object including an attached prototype chain. By default all created scope objects only have the special object `Object.prototype` as prototype, but via features like `with` and direct access to the scope chain, more complex structures can be created. Injected code could potentially abuse this to alter the code of an intended harmless program.

When a variable is accessed, JavaScript first searches in the most recent scope object and its prototype chain. If the variable was not found, the search continues in the prototype chain of the next more general scope object. Prototype chains of the scope objects might share objects or even link to other objects included in the scope chain.



Every function call adds a freshly created scope object to the scope chain in which the function is defined. The local variables of the function body are stored in this fresh scope object and the function body has furthermore access to all variables defined in the scope of the function definition. In contrast to other popular languages like C or Java, blocks {...}, including the branches of a conditional statement or the body of a while loop, do not create a new scope object but are ignored for the bounds of the scope.

JavaScript programs can manipulate the scope chain directly. By using the `with` statement, an expression can add an arbitrary object `o` as most local scope to the scope chain. As a result, local variables in the newly defined scope are equivalent to fields of `o`. Adding or deleting fields of `o` also manipulates the available local variables. By other methods, a JavaScript program can obtain a reference to the global scope object. Modifications of this object change the global variables equivalently. This behaviour, called *dynamic scope*, results in a scope structure depending on the runtime state of the program. Depending on the current runtime values, a variable `x` might address different locations in the scope chain.

Another unintuitive feature of JavaScript are the declarations. Local variables and function statements declared throughout the scope's code are moved to the top of the scope. This behaviour is called *hoisting*. JavaScript executes the code of a scope in three passes. In the first two passes only function statements and local variable declarations are inserted into the fresh scope object. During this phase, declarations are executed independent of control flow operators. For example, declarations in both branches of an `if` statement are executed. The final pass executes the actual code in this prepared scope with all the local variables already declared.

The interaction between hoisting and JavaScript's scoping sometimes results in surprising behaviours. In the example in Figure 2.1, the outer scope defines a variable `x`. The call to function `f` in line 6 creates a new scope objects on the scope chain and initialises it with the variable declarations in the function body. Therefore, the scope object for the function body already contains the local variable `x` before the function body is executed. The assignment in line 3 assigns the value 2 to the variable `x` in the inner scope and the value of the variable `x` in the outer scope still equals 1 after the function call.

If a JavaScript file is executed in strict mode, the `with` operator and similar features are not available and the program is instead statically scoped: the structure of the scope is independent of the runtime state, but instead only depends on the position of the

Figure 2.1 Example Hoisting

```
1  var x = 1;
2  function f() {
3    x=2;
4    var x;  // Executed before the assignment in line 3
5  }
6  f();
7  x; // equals 1
```

executed expression in the code. This simplifies the situation for analysis.

2.1.1.3 Hidden memory operations

An assignment to an undefined variable creates the assigned variable fresh in the most global scope object without warning or error. Object extension behaves similarly. If an expression assigns a value to a new field of an object, this field is silently created in the given object. This implicit definition poses a challenge to code analysis, since the set of defined variables or the set of fields of an object is difficult to determine.

In the example in Figure 2.2, line 2 implicitly declares the global variable `x` by assignment. Line 3 extends the object stored in `x` by the field `m`. After the function call the field `x.m` contains the value 3, even though the variable `x` and the field `m` were never declared.

Figure 2.2 Example Implicit Allocations

```
1  function f() {
2    x = {};
3    x.m = 3;
4  }
5  f();
6  x.m; // equals 3
```

2.1.1.4 Flexible function definitions

JavaScript offers polymorphic and variadic functions. *Polymorphic* functions can be called with parameters of different types, while *variadic* functions can be called with a variable number of parameters. If a function is called with fewer parameters than specified in the function definition, the remaining parameters store the value `undefined`.

If the function call provides more parameters than specified in the function definition, the additional parameters are accessible through the `arguments` object. In the example in Figure 2.3, the function `f` is defined with exactly 1 parameter but called with 2 parameters in line 1 and without parameters in line 9. When the function is called without parameters the explicitly defined parameter `x` is set to `undefined` as used in line 3. Line 5 and 7 access the non-specified parameter via the array `arguments`. This way, the function can return the input `6` for the call in line 1 without naming it in the function definition.

Figure 2.3 Example Flexible functions

```
1  f(5,6);   //returns 6
2  function f(x) {
3    if (x==undefined)
4      return 0;
5    if (arguments[1] == undefined)
6      return 1;
7    return arguments[1]
8  };
9  f();      //returns 0
```

JavaScript functions fulfil three different roles: functions, methods and constructors. The subtle difference is the value of the variable `this` inside the function body. Consider Figure 2.4. A *method* (line 5) is called as a field of an object and the variable `this` contains the receiving object. A *constructor* (line 6) is called with the prepended `new` operator and `this` contains a newly created empty object to be extended by the constructor body. A *regular function call* (line 7) is called without additional keywords and `this` is bound to the global scope object.

Figure 2.4 Example Functions, Methods, Constructors

```
1  function f() {
2      return this;
3  }
4  var o = {m:f};
5  o.m();      //returns o
6  new f();    //returns a fresh object {}
7  f();        //returns the global scope object
```

2.1.1.5 Dynamic types

During runtime, values in JavaScript have dynamic types associated with them. However, value updates are not restricted by the type. An integer variable can be updated with a string or object value, changing its dynamic type. This behaviour may result in dynamic variable types dependent on the previous control flow. In Figure 2.5 the local variable `y` is initialised as object in line 2. Line 3-6 update `y` either with a string or an integer depending on the value of `x`. Therefore, the resulting type of `y` in line 7 depends on the value of `x`.

Figure 2.5 Example Dynamic Types

```
1  function f(x) {
2    var y = {};
3    if (x==3)
4      y = ''String'';
5    else
6      y = 4;
7    return y;
8  }
```

2.1.1.6 Coercion

As scripting language, JavaScript is designed to throw as few exceptions as possible. For this purpose, JavaScript converts many values automatically into the required dynamic type. For example, the `if` statement expects a Boolean as condition but an integer or even string value as condition will be converted to Boolean by a list of rules. One extreme example are the operators + and ==. The + operator first converts both operands to primitive values via the `valueOf` method. It then executes string concatenation if the typeof function identifies at least one of the primitive operands as String. The other operand is potentially converted to String via the `toString` method. Otherwise, both operands are converted to a numeric value and arithmetically added. The == operator has 9 separate cases which involve converting Objects and Function to primitive values and Boolean and String values to numeric values. Only in cases where these extensive coercion protocols fail, mainly when accessing `undefined` as object or function, a TypeError is thrown.

Conversions from primitive values to objects produce object wrappers, which contain the original value and can be extended with additional fields. It is worth noting

that, if a variable gets wrapped in this way, the value of the variable in the scope is not overwritten with the wrapped object. This can lead to surprising behaviour. Consider, for example, the code in Figure 2.6. Line 1 defines `x` to be an integer. Line 2 accesses the field `a` of `x`. Since integers do not have fields, JavaScript reads the value of `x` from the scope, creates an integer object wrapper and assigns to its field `a`. Since the result is not assigned back to `x`, the original variable is still a simple integer. The second conversion in line 3 creates a new integer object wrapper without a field `a` and therefore the return value is `undefined`. Line 4 converts `x.a` to a Boolean. Since the value `undefined` converts to `false`, line 5 is not executed.

Figure 2.6 Example implicit Conversion

```
1    var x = 4;
2    x.a = true;
3    x.a; //equals undefined
4    if (x.a) //undefined converts to false
5      x = 6;
```

2.1.1.7 Metaprogramming

JavaScript's `eval` operator takes a string and executes it as JavaScript code. In practice, evaluated strings are often constructed dynamically or even received from a remote server. As `eval` can parse the whole JavaScript language, its arbitrary effect is difficult for static analysis. Figure 2.7 shows a program which downloads a string from a remote server and executes it as JavaScript. The behaviour of this code cannot be determined before execution. Static analysis has to assume the worst-case. However, previous research [99] has shown that `eval` is often used for operations which can be achieved without the use of `eval`. For example, JSON data obtained from a server can be parsed using the JSON library instead. Based on pattern recognition, many use cases for `eval` can therefore be translated into analysable code.

2.1.1.8 Computed access

JavaScript provides two different modes to access object fields. The static access `o.m` explicitly uses the identifier of the field `m`. The computed access `o[`e_1`]` in contrast allows for the name of the field to be computed by an arbitrary JavaScript expression e_1. The result of the expression is converted to a string and then used as the identifier

Figure 2.7 Example Arbitrary Code Execution

```
1  xmlhttp=new XMLHttpRequest()
2  xmlhttp.open("GET","target.com",false);
3  xmlhttp.onreadystatechange = function() {
4    eval(xmlhttp.responseText);   //execute received command
5  }
6  xmlhttp.send()
```

of the field. Usually, during static analysis the exact result of the expression e_1 is not available. Therefore, a sound type system has to make conservative assumptions about the result of e_1 and potentially assume every field of the object has been accessed.

2.1.1.9 Interpretation and extension of the standard

Although JavaScript is standardised by ECMA, implementations of JavaScript interpreters are diverse. Some implementations include extra features which are not specified by the standard. For example, the `window` variable in most browsers always points to the global scope object. This feature is not part of the ECMA standard. Another example is the direct manipulation of the prototype chain via the field `__proto__` of an object. On the other hand, implementations lag standardised features or alter their behaviour. At time of writing, according to `http://kangax.github.io/compat-table` none of the popular browsers were 100% compatible with the standard ES5. For example function statements declared in both conditional branches as shown in figure 2.8 behave differently depending on the browser.

Figure 2.8 Example Differences in JavaScript implementations

```
1  function f(x) {
2    if (x==3)
3      function g() {return 1;}
4    else
5      function g() {return 2;}
6    g() // might be 1 or 2 depending on the browser
7  }
```

2.1.2 Formal JavaScript semantics

Various formalisations of the JavaScript semantics have been proposed covering different aspects of the language.

For the version ES4 of the JavaScript standard (which was never formally released) a reference implementation in ML covering the whole language was planned [53]. This implementation covers all ES4 core features including the planned gradual typing. However, since ES5 and ES6 build upon ES3 rather than ES4, this implementation cannot be used for real-world code.

A core of ES3 has been formalised as λ_{JS} [49]. This work provides a big-step evaluation semantic which mainly focuses on the object behaviour of JavaScript including prototypes. The remainder of the JavaScript language, which λ_{JS} does not cover, is translated into the core language by a desugar function. More details on this work are given in Section 5.4.5.

Maffeis, Mitchell and Taly [81] aim to formalise the whole standard ES3. This includes the scope chain built from objects, the proper handling of hoisting, redefinition of JavaScript built-in features and the correct binding for `this` in functions. The presented semantics has a modular design to account for differences in JavaScript implementations. Some JavaScript features are omitted, most importantly the `switch` and `for` statements, the `Date` and `Math` libraries and regular expression matching.

SES [2] is a semantics presented by Agten et al. describing a subset of ES5S, which can be handled easier than the whole of ES5S. Taly [111] extends SES to SES_{light}, which covers almost full ES5. The formalised language lacks getter and setter operators, restricts writes and extensions of built-in objects, does not handle prototypes within the scope chain and requires the JavaScript features executing dynamic code (`eval` and computed field access) to specify a set of variables, which the dynamically executed expression might modify.

Most of ES5 is captured by the semantic JSCert [19] formalised in Coq. From this formalisation the authors derive the OCaml implementation JSRef which evaluates real-world JavaScript code and is proven sound in relation to JSCert. Via the implementation JSRef, JSCert can be tested against the JavaScript reference tests. This formalisation only lacks dynamic JavaScript parsing (needed to handle the `eval` command) and most of the built-in libraries like `Math` or regular expressions.

The system KJS [92] models the whole of JavaScript version 5.1 (ES5.1) and a major part of the libraries for Objects, Functions, Booleans, Errors, Arrays, Strings and

Numbers. The evaluation of the semantics passes all reference test of the JavaScript specification. In the absence of a formal specification for JavaScript, this semantics can be used as a reference implementation.

2.1.3 JavaScript in the real world

Ultimately, research on JavaScript has to be evaluated on code used in the real world. However, most research makes assumptions about JavaScript code. The validity of these are evaluated in various studies. This includes checking the differences between theoretic systems and JavaScript implementations as well as the differences between benchmarks and real-world JavaScript applications.

Pradel et al. [95] study type coercions in JavaScript code. They instrument all operations which could lead to coercion and dynamically extract coercions that happen in a typical run. They find that coercions are widely used and classify most of them as harmless. Many occur in specific expressions, for example converting the result of a conditional expression to Boolean or initialising a variable with a default value of a different type than the actual value. This work also checks for the use of strict and non-strict equality (=== against ==) and finds that they are used interchangably although in most cases strict (i.e. without automatic coercion) is sufficient and probably intended.

Richards et al. [101] inspect the traces of benchmarks and popular web pages during meaningful use. They find that polymorphic and variadic functions cannot be neglected and programs widely use prototypes, object extensions and dynamic features like `eval` and computed object lookup. The comparison between benchmarks and real-world code shows that the runtime of benchmarks is too short and they do not use enough `eval`, polymorphism of constructors and object modifications to represent real-world programs. JSMeter [96] also compares 11 popular web pages with the V8 benchmarks and selected parts of the SunSpider benchmark collection. They discover that the considered benchmarks do not test the complex use cases used in the real-world JavaScript code. Therefore, tests on the benchmarks might be misleading.

Previous work also surveyed the way `eval` is used in existing code. Richards et al. [99] find that `eval` is not solely used for parsing JSON code, but that there are multiple patterns which it is used for. Jensen et al. [63] go a step further and propose the Unevalizer framework which replaces `eval` uses with constant arguments, JSON or other simple code strings with alternatives using dataflow analysis. Their proof-of-concept implementation can handle 75% of the `eval` calls in the experiments. The tool

Evalorizer [83] has a similar goal, but classifies the different uses of `eval` dynamically using execution logs. With this technique the authors are able to replace 97% of the calls.

Similarly, Park et al. [91] try to rewrite the uses of the `with` statement. They discover 7 different patterns for the use of the `with` command in real-world code and propose automatic rewriting strategies for all of them. In an evaluation they are able to rewrite all uses of `with` except for those mixed with dynamic code execution. These rewriting techniques can be used to transform real-world JavaScript applications into analysable code.

For practical JavaScript programming Crockford [25] describes a subset of JavaScript which should be used since it is free of confusing ambiguities. He discusses how known programming paradigms can be achieved in JavaScript and how complex functionality can be programmed without error prone functions like `eval`. However, this work is not formalised.

2.1.4 JavaScript for mobile phones

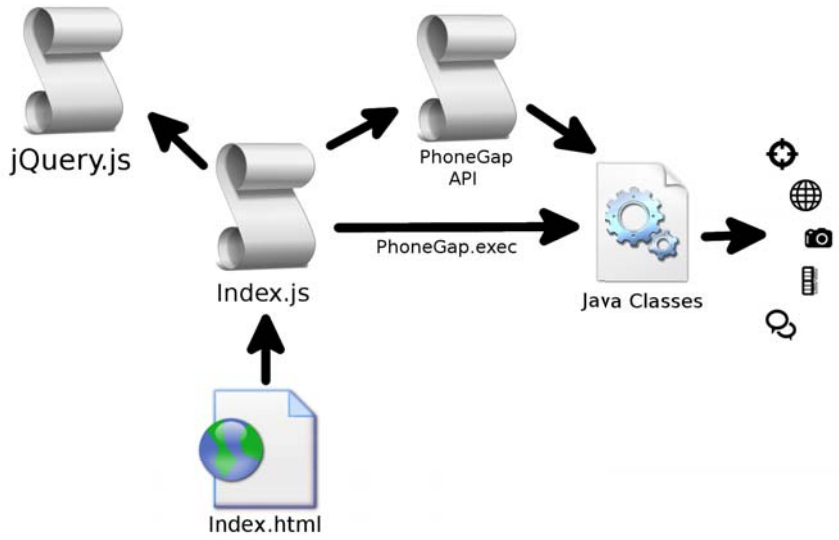
Most current mobile phones have access to a constant Internet connection and many existing mobile apps provide features similar to web applications. The phones already provide a JavaScript interpreter as part of their browser. Therefore, JavaScript is an obvious choice as programming language for mobile apps. New emerging mobile systems build directly upon JavaScript, for example Samsung's Tizen OS [38] or Firefox OS [90] and its derivative H5OS [112]. Apple provides the JavaScriptCore framework to call JavaScript programs from native code and the Ubuntu phone OS [79] advertises its HTML/JavaScript SDK.

For all established conventional systems - including Android, iOS, Blackberry and Windows - various frameworks provide an easy way to develop apps in JavaScript.Figure 2.10 surveys a selection of the popular such frameworks. An accurate survey about the available apps and the users of the frameworks in the real world would require the download of a huge amount of apps directly from the Google Play Store. Since this is not feasible due to Google's Terms of Service, Figure 2.10 approximates the number of available apps by the apps showcased on the framework's web page and the number of users by the registered users in the development forums. This approximation should give an overview of the reach of the different frameworks.

This work will focus on PhoneGap, one of the most popular such frameworks, as

a typical example to create JavaScript apps. A JavaScript app written with PhoneGap consists of 2 different parts: the platform dependent *native part* is compiled into the native language of the operating system and has direct access to the phone's services and resources. The *JavaScript part* implements the user interface and functionality of the app with access to the platform independent API of the native part.

Figure 2.9 PhoneGap structure



At launch the operating system executes the native part. It creates a browser view in the respective operating system, initialises the PhoneGap APIs and executes the HTML/JavaScript part (index.html) within this browser view. PhoneGap developers have access to the full JavaScript features, including third party libraries like jQuery, as the app is executed in a standard browser. Additionally, the framework provides a bridge to access the APIs of the native part: the function `exec` is available within the JavaScript part and can call methods of the native part.

This `exec` bridge is usually used to implement plugins. Each plugin provides a Java class with the native code to access a specific resource and a JavaScript API mapping the JavaScript calls to the Java class methods via the `exec` bridge. A set of core plugins to access the most common resources is provided by PhoneGap directly and further functionality like access to various mobile phone sensors is accessible through plugins maintained by third party developers and listed on the PhoneGap web page.

The back-end of PhoneGap is the Apache Cordova library, which provides the bridge and essential functionality for the app's life-cycle. Both PhoneGap and Cordova are available in Version 5.3.1 via the npm repositories at the time of writing.

The advantage of PhoneGap apps over pure native code is the portability: the JavaScript code of the app can be reused and packaged for different platforms by replacing only the native part of the plugins. The user interface code can be executed and tested in every browser which makes development and preview easy and the developer has access to the large set of available JavaScript libraries and frameworks which improve code development (e.g. *jQuery*), enhance maintainability (e.g. *react*) or provide powerful user interfaces (e.g. *enyo*).

Figure 2.10 JavaScript frameworks for mobile apps

Framework	URL	Released	Apps[1]	Users[2]	Functionality
PhoneGap	`phonegap.com`	2009	3900[3]	24,000[3]	developed the Cordova library
Sencha Touch	`sencha.com/products/touch`	2010	3	534	commercial model-view-controller paradigm optimised UI elements compiles Java code into JavaScript deploys via PhoneGap
ratchet	`goratchet.com`	2012	3	318	prototyping JavaScript UIs deploys via different frameworks
Ionic	`ionicframework.com`	2013	120[4]	4400	commercial execution in browser desktop and mobile
tabris.js	`tabrisjs.com`	2014	6	N/A	front-end for Cordova maps HTML to native UI elements
react native	`facebook.github.io` `/react-native`	2015	90	380	maps HTML to native UI elements integrates with native code
NativeScript	`nativescript.org`	2015	12	363	open source compile JavaScript into native

(all numbers as of December 2015)

[1] as showcased on the page
[2] active on the support forum
[3] stating 400,000 developers and 1 million apps on their web page
[4] stating 875,000 users on their web page

2.2 Resources for mobile devices

JavaScript accesses two principal kinds of resources. One set of resources, the *language activated* resources, is consumed by the operators of the language directly. Those resources include memory space, processing time and power. The second type of resources, the *API activated* resource, is consumed by *invocation of APIs*, for example any of the PhoneGap plugins mentioned in Section 2.2.2. In mobile apps this category of resources can be further classified as the private data, services (SMS, phone calls or push notifications) and sensors (camera and microphone, GPS, accelerometer and gyroscope).

Both language and API activated resources can be abused by attackers. One obvious attack is violation of privacy. The GPS sensor reveals the user's location and movement patterns and access to the microphone and camera can potentially record

confidential information. Overuse of memory, time and power can lead to denial-of-service attacks, since the resource is not available for other apps. In the smartphone case this also affects the user's ability to start and receive phone calls. Notifications are also targeted by denial-of-service attacks. If an app succeeds to flood the notification bar with unimportant notifications, potential security warnings might stay unnoticed [20]. This can turn into a social engineering attack, in which the app repeatedly displays less important confirmations to hide one critical request. By the time of the serious request the user's attention might have dropped enough to confirm the important request like an unimportant one. Furthermore, the smartphone can execute billable services like SMS and phone-calls. Attackers can send SMS to premium numbers billing into their bank account and thus cause unexpected bills.

The different kinds of resources result in different challenges to provide bounds on the usage. The same API activated resource can be consumed by multiple API methods. To guard an API activated resource, all access methods need to be considered. On the other hand, analysis of language activated resources needs to take into account the fine details of the JavaScript semantics, which can even vary between different JavaScript implementations. Concerning language activated resources, this thesis only considers memory space, since compared to time and power consumption, memory consumption is less dependent on the device and subtle differences in the machine state. State-of-the-art systems [88] in this research area estimate power consumption of apps with power models with various parameters including the device components (display type) and settings (brightness).

2.2.1 Classification

Independent of the differentiation between language and API activated resources, resources can also be classified in respect of their consumption behaviour. The following two questions characterise the consumption behaviour of a resource kind:

- Does an allocation of this resource acquire exactly one unit (unitary) or is it possible to acquire units of this resource "in bulk"?
- Is it consumed irreversibly (consumables) or can the resource be released again?

All combinations of those two properties results in 4 categories. Each category raises different analysis questions, depending on how resources in this category are used or abused.

count: Every resource usage is unitary and the resource is consumed on usage. For this resource the consumption is equal to the number of accesses. For example, the number of HTTP requests sent or the contacts read are both resources in this category. The analysis question in this context is "How often did the app access this resource?" and aims to protect e.g. against unreasonable leakage of data.

cumulative: Multiple units per access are possible, acquired units get consumed. An example is the data traffic or carrier credit used. The analysis answers the question "How much of this resource has been consumed in total?"

unique (blocking / non-blocking): Only one unit can be acquired, but the unit can be released after use. Apps ask (explicitly or implicitly via a library call) for the access to this resource and ideally free the resource after the resource operation has been performed. This could be blocking (i.e. only one app can have this permission at any one moment) or non-blocking (multiple parallel apps allowed). The user is concerned about the questions "Does the app currently hold this resource?" and "Did the app ever hold this resource?" Analysis of the first question protects against monopolies and deadlocks and the second can be used to infer access to critical data and sections like the password store.

acquire-release: A program can acquire a certain amount of units of this resource and return parts or all of them after use. The analysis questions are "How many units were held maximally?" and "How many units have never been returned after execution?" Examples in this category include the memory space.

2.2.2 Resources in PhoneGap

The strength of PhoneGap apps over JavaScript in a web page is PhoneGap's infrastructure for plugins to access the native resources of the mobile phone. On Android these plugins are implemented as Java class and provide a JavaScript API to invoke the Java methods. The central registry for PhoneGap lists 1113 plugins [22nd September 2015] and the npm repository lists 552 [22nd September 2015] for the latest version of the framework. 18 of them are published as core PhoneGap plugins on the PhoneGap webpage [59]. The others have been developed by third party developers. The following shows how each of the mobile phone resources is accessed in PhoneGap.

Camera The camera is accessed using the PhoneGap plugin `cordova-plugin-camera` with its function `navigator.camera.getPicture`.

This API activates the camera app of the operating system, inviting the user to take a picture, which is then returned to the app. Since it does not take a picture directly, it does not require the permission to access the camera. The function `navigator.device.capture.captureImage` implemented in the plugin `cordova-plugin-media-capture` provides similar functionality and can additionally capture a video in a similar way via `navigator.device.capture.captureVideo`. Even though these APIs do not access the camera without user interaction, a limit on the times an app is allowed to call these APIs can prevent the app from flooding the user with requests to take a picture.

Location The plugin `cordova-plugin-geolocation` provides two methods to access the location: `navigator.geolocation.getCurrentPosition` queries the location once, whereas `navigator.geolocation.watchPosition` returns the location repeatedly. Instead of PhoneGap's bridge to Java, the newest implementation on Android directly uses the W3C Geolocation API specification of the browser view. In older versions of this plugin, the API functions were called `getlocation` and `addWatch`. Any of these APIs require the permissions `android.permission.ACCESS_COARSE_LOCATION` or `android.permission.ACCESS_FINE_LOCATION`. Location data could also be obtained through the EXIF data extracted from pictures. Therefore, the method `navigator.camera.getPicture` should to be considered for location information, too.

Messaging PhoneGap does not promote an official SMS messaging plugin. Instead, a range of 3rd party plugins with different APIs are available. In the 8757 apps available for this evaluation 19 different messaging plugins were found and further 5 are listed in the PhoneGap plugin repository. Out of these 24 plugins, 12 are only included in one specific app and not available as source code. The APIs of the available 12 plugins are summarised in Figure 2.11. All analysed plugins provide exactly one API function each to send SMS with a specified text to the specified recipients. Additionally, some plugins provide functions to check whether the current phone has the capability so send SMS or to receive messages from the user's inbox. While reading SMS could be considered as part of the "private information" resource, the obvious resource for those plugins are the sent SMS.

Instead of sending messages directly, apps can trigger the user's chosen messag-

ing app via intends. In this case the message is send by the messaging app and the PhoneGap app does not require a permission. There are various third party plugins with APIs to send messaging intends (some included above) or intends in general. Since the user has to specifically press the familiar "Send" button, this method is not as critical for the resource analysis.

Movement To access the accelerometer and compass, PhoneGap provides the plugins `cordova-plugin-device-orientation` and `cordova-plugin-device-motion`. The accelerometer is accessible without permission, while the compass requires the same permissions as the location. Both plugins provide a method `navigator.accelerometer.getCurrentAcceleration`/`navigator.compass.getCurrentHeading` to access the sensor once and `navigator.accelerometer.watchAcceleration`/`navigator.compass.watchHeading` to access the sensor repeatedly.

Notifications The main notification plugin in PhoneGap is `cordova-plugin-dialogs`. It does not require additional permissions. This plugin implements the object `navigator.notification` with the 3 methods `alert`, `confirm` and `prompt`, which each display an overlay window with different configuration of buttons. Additionally, it provides the method `beep` for audio notifications. JavaScript directly provides similar functionality with the built-in functions `alert`, `confirm` and `prompt`.

The plugin `cordova-plugin-vibration` requires the permission `android.permission.VIBRATE` to activate the rumble feedback of the device. With this additional plugin the app might use the methods `navigator.vibrate` or `navigator.notification.vibrate` and `navigator.notification.vibrateWithPattern` for tactile feedback.

The notification area in the status bar can be manipulated with the plugin `cordova-plugin-statusbar` without further permissions. It implements the function `StatusBar.hide` and `StatusBar.show` to change the visibility of the system's status bar.

Contacts The contacts stored on the phone are guarded by the permissions `android.permission.WRITE_CONTACTS` and `android.permission.READ_CONTACTS`. In PhoneGap they are accessed through the plugin `cordova-plugin-contacts`,

which provides the API object `navigator.contacts`. Its method `pickContact` display the system's "choose a contact" dialog and `find` can access the contacts without user interaction. With the function `create` a new contact object is created. Each contact object (freshly created or obtained from the existing database) has the method `save` and `remove` to alter the database permanently.

Audio Much like the plugins to take pictures, the plugin `cordova-plugin-media-capture` can use the function `navigator.device.capture.captureAudio` to launch the sound recorder of the operating system. In contrast, granted the permissions `RECORD_AUDIO` the plugin `cordova-plugin-media` can directly access the microphone via the function `media.startRecord` and `media.stopRecord`. It can play audio and video files using the function `media.play` and change the volume via `media.setVolume` with the permission `MODIFY_AUDIO_STATE`.

Files The plugins `cordova-plugin-file` and `cordova-plugin-file-transfer` operate on the file system. They provide various functions to write and read files, but are usually restricted to write to the app specific storage. Additionally the `FileSaver` object can monitor file changes.

Other private Information The plugin `cordova-plugin-battery-status` provides the events `lowbattery` and `chargerconnected` to react to the **battery state**. The plugin `cordova-plugin-device` accesses **device information** like the device model, its Universal Unique Identifier and the version of the running OS. The plugin `cordova-plugin-network-information` allows to inquire the status of the network **connection** via the method `navigator.connection.type`. Furthermore, it provides the events `offline` and `online` to react to changes of this status. The required permissions `android.permission.ACCESS_NETWORK_STATE` and `android.permission.ACCESS_WIFI_STATE` are requested by the PhoneGap framework already. The plugin `cordova-plugin-globalization` provides several methods which return **location-settings** like the preferred language, localisation scheme and formats for numbers, dates and currencies without further required permissions.

In general, the plugins included in the PhoneGap plugin database are maintained by various third party developers and come without any guarantee. However, emerging databases like the Telerik [24] marketplace collect open-source plugins and verify them

manually. Their aim is to provide a guidance for developers, but the documentation resulting from this process also lists all information needed to identify the resource consuming APIs for each plugin.

2.2.3 Resource policies for mobile devices

2.2.3.1 Policy models for mobile phones

Most current mobile operating systems employ a *permission-based* policy to guard the resources. Apps request permissions for each required resource. The user has to grant all requested permissions to install the app. After this initial request the app can use the requested feature freely without additional user interaction. Previous research [37] suggests that users do not understand the implications of the permissions or do not consider the permissions granted during install time carefully enough. Newest versions of the Android OS have added options to grant or reject single permissions during runtime. Even with this modification each permission is either denied or granted unconditionally, independent of the quantity and context in which the resource is accessed. This information can be the difference between wanted and unwanted behaviour: a messaging app has to send SMS to the provided numbers, but only one per recipient and not without user interaction or to premium numbers. A voice recorder should have access to the microphone, but only after the user pressed the recording button.

Capabilities [76] offer a finer access control model than permissions. A capability is an unforgeable token that allow access to a resource. These tokens are provided, for example as parameters to a function, to the parts of the program that legitimately requires access and can be forwarded to the appropriate subfunctions to delegate access. In some capability systems such delegation is restricted by an additional policy. A capability can grant access to a single file or enable access to a whole set of resource APIs.

Confirmation-based policies involve the user in the policy decision during runtime. The user has to approve every resource access separately. This system might fit more important resources but quickly drains the user's attention if too many requests have to be confirmed. If the mental capacity of the user is exhausted, important confirmations hidden in a set of less important confirmations have a good chance of being overlooked. Confirmation based policies also have to handle re-requests appropriately since otherwise the app could simply ask for the permission repeatedly until it is granted. The option to grant access in a bulk [13] can improve this approach, for example, granting

the next 10 accesses or all following requests of this app. This offers a pay-off between fine-grained control over the resource and mental load. Another important addition to confirmations for resources, as discussed by Ben Poiesz in the Google I/O 2015, are reasons why the permission is requested. If the app provides such a reason, the user can make an informed decision whether to grant the request.

The approach taken in this work aims to combine the permissions based approach with advantages of "in-bulk" confirmations. The aim is to provide for each granted permission the information how often and in which context it is used. This information helps to differentiate between intended features and unwanted behaviour.

This is achieved by providing bounds on the consumption of each resource. These bounds depend on the runtime situation of the app: the bound on the consumption of a function depends on the size of the input data and the bound for the whole app depends on the user interaction. The resulting bounds can be described as *ticket-based* policies: instead of the permission for unrestricted access to a resource, the app receives a number of tickets. Every time the app requests access to the resource, it has to "pay" one ticket. For data-dependent bounds, these tickets can be stored inside the data structures. For example one SMS ticket can be stored with each contact in a recipients list. For interaction-dependent bounds the tickets can be generated by the interaction event. Each press to the button "Send" generates a fresh ticket to send exactly one SMS. This way, the press to the "Send" button acts as implicit confirmation by the user, embedded into the work-flow of the app without consuming the user's mental capacity. These policies are formally introduced in Chapter 3 and implemented by the systems in Chapter 4 and 6.

2.2.3.2 Policy enforcement systems

Previous studies [34, 72] have shown that users typically do not know what each Android permission implies. Most users have accepted that standard permissions, such as full Internet access, are requested by almost all apps and do not pay attention to the permissions requested by an app during installation. The system Kirin [31] defines rules describing which sets of permissions collude to a potential thread and warns the user during installation.

The tool Dr. Android and Mr. Hide [66] divides the existing Android permissions into finer sub-permissions and enforces their usage by a dynamic monitor like PhoneWrap. The enforced policies are more fine-grained than the stock Android policies, but neither interaction-dependent nor quantitative.

Jin et al. [68] enforce more fine-grained access control specifically for HTML5 and PhoneGap apps. Their system allows to grant permissions for each iFrame of the app separately. This way, they can grant access to the UI of the app but restrict views inserted by third parties, e.g. ads. The policies presented in this thesis are even more fine-grained, since they can grant or deny access to each button separately.

There are also various systems which enable the user to control the permissions with more granularity: MockDroid by Bereford et al. [17] allows the user to revoke permissions to each app during runtime, causing the operating system to report the resource as unavailable or to return empty data. This is achieved by modifying the Android access control system directly. The system CRéPE by Conti et al. [23] also modifies Android and allows the user to restrict resources to a specific context based, for example, on time or GPS location. The system denies all access outside the specified context. The system PhoneWrap presented here can replace resource access with the same countermeasures as MockDroid and allows more fine-grained context-dependent control than CRéPE based on user interactions with the app.

Felt et al. [35] discuss different methods to make the user of an app aware of resource consuming actions with the conclusion that the use of warnings and confirmation dialogs should be minimised to reduce the habituation effect. The interaction-dependent policies discussed in this thesis implement the “Trusted UI” paradigm presented in this paper. Enck et al. [30] confirm this paradigm as one of the programming guidelines for smartphone security.

Access Control Gadgets [102] implement resource policies similar to the interaction policies in PhoneWrap. However, the UI elements granting access in their approach are supplied by the resource APIs and therefore not as flexible. PhoneWrap policies instead can choose any UI element from the original app making the application of PhoneWrap to existing apps easier. Furthermore, their approach needs to modify the execution environment to capture the events and protect the gadgets. PhoneWrap achieves this without modifying the execution environment using JavaScript’s inherent features.

In an evaluation on over 4000 apps, Elish et al. [29] determine that user interaction is a good metric to distinguish between functionality and suspicious behaviour. They examine the portion of critical API calls in the tested apps triggered by user interaction and discover that the pattern is significantly different for malware and benign apps. The interaction-dependent policies described in the latter chapters of this thesis use exactly this difference to restrict unwanted behaviour.

Quantitative resource policies have been applied to Java apps [26, 33, 108]. Since Java apps are usually compiled, it is more complicated to wrap the resource APIs with access to the user interface. Especially PhoneWrap lightweight approach to implement the resource monitor in the target language is not possible here. Therefore, the existing tools have to make more complicated modifications.

Wrapping methods injected into pure Java apps have also been considered. AppGuard [15] disassembles an App and inserts inline reference monitors to mediate the granted permissions with more fine-grained control. Aurasium [118] replaces the libc library included in the app to mediate resource accesses. AppGuard could potentially be extended to incorporate user interaction into its policies, but in the current state neither AppGuard not Aurasium enforce interaction-dependent policies. However, since PhoneGap apps handle all user interaction in the app layer implemented in JavaScript, an enforcement of interaction-dependent policies for PhoneGap apps has to be implemented on the JavaScript layer.

Ticket based policies are discussed by Besson et al. [18] and further examined by Aspinall et al. [13, 14]. The presented system allows to approve a number of resource accesses in one confirmation dialog. A dynamic enforcement ensures that an action is only performed if the program is granted sufficient access to all required resources. The authors then show that under certain conditions the runtime checks used to enforce the policies can be omitted.

Despite the attempts in [6] to initiate an adaptation of the mobile phone permissions to web apps, there is only a sparse protection of critical functions in browsers so far. The PhoneWrap policies can be applied to web pages to guard the browser-specific resources including the URL bar, which malign scripts should not write into, pop-up windows and cookies. However, web pages are downloaded fresh every time the user visits or navigates. That means, the injection of the enforcement has to be performed for each page load. The browser Opera and its built-in feature `UserScripts` enables PhoneWrap to inject the wrapping script automatically into every fresh loaded page and enforce PhoneWrap policies for web pages.

2.3 Analysis of JavaScript programs

Previous research of program analysis has taken two different principal approaches: static or dynamic. In *static* analysis the source code or intermediate representation is analysed without executing it. The results over-approximate all possible executions

of the analysed program. This might also include branches which are never executed. Static analysis is often executed on a device different than the target device of the program and runtime overhead is not as critical since the device executing the analysis can be much more powerful than the execution device.

The *dynamic* approach monitors one specific execution of the program during runtime. Therefore, the result only describes one branch of the program with more precision than static analysis. Dynamic analysis omits dead code, but changes the runtime performance and potentially the behaviour of the app.

The following presents existing systems and their properties. The resource analysis presented in later chapters builds upon the existing methods and assumptions.

2.3.1 Dynamic analysis

The work on PhoneWrap is inspired by self-protecting JavaScript [94], which introduces the framework to wrap JavaScript objects and enforces state-based policies in web applications. The work in Chapter 4 extends this work to JavaScript-based mobile phone apps and enforces concrete interaction-dependent policies. Magazinius et al. improved the original wrapping method in [82] to protect the JavaScript internal methods against re-definition. The improvements made there apply in the same way to the system PhoneWrap.

A common approach is to modify of the browser environment to dynamically enforce security properties of JavaScript applications. ConScript [84] modifies Internet Explorer 8 and provides a framework to enforce fine-grained policies in web applications. Example policies include restrictions of the JavaScript capabilities of included third party scripts or communication monitoring. Different systems [27, 46, 93] enforce information flow policies by modifying the Firefox browser: jcshaddow [93] implements separated variable stores for JavaScript scripts from different sources to minimise leakage between different scripts, Adsentry [27] executes every script in a different JavaScript engine and Groef et al. [46] multi-execute one script on two levels, once with private knowledge but no outputs and once with output but without privileged data. This guarantees that the output does not leak private data. Richards et al. [100] modify the WebKit engine to annotate JavaScript functions and values with ownership and enforce fine-grained access rights. All these system rely on the modification of the environment and are therefore specific to one exact version of a browser. PhoneWrap, in contrast, inserts the enforcement into the app package and can be ex-

ecuted on any system which can execute the original app. In addition, none of the systems above is capable of enforcing interaction-dependent policies.

Another dynamic approach is to modify the source code of the web page. AdJail [113] modifies the code on a shadow-server to separate all scripts from the main page using iframes. To preserve essential operations AdJail allows communication between scripts specified in a white list. SafeScript [78] rewrites JavaScript code to isolate different scripts. Barth et al. [16] rewrite JavaScript files on a proxy to replace insecure function calls with equivalent secure ones. None of the rewriting systems is able to enforce policies dependent of the user interaction. Furthermore, while rewriting is similar to the wrapping performed by PhoneWrap, rewriting needs to take extra care that dynamically injected code is rewritten as well. PhoneWrap's wrapping instruments the functions defined in the JavaScript environment which automatically affects all dynamically inserted code and therefore exposes less attack surface than rewriting. However, all methods used above to rewrite could be used to insert the PhoneWrap wrapping script into web-pages.

The system BrowserShield [97] can enforce policies similar to PhoneWrap including the interaction-dependent policies. However, this system heavily modifies the browser and is thus not portable to mobile phones or between different platforms. Modifying the browser view of PhoneGap apps in this way would require root access to the mobile phone, which many users are not able or willing to activate.

2.3.2 Static analysis

Apart from type systems, which will be discussed in more detail in Chapter 5, there are various approaches to static analysis for JavaScript.

Gardner et al. [42] present a logic to describe and reason about complex properties of the whole JavaScript language. The fully formal approach to the whole language makes automatic inference hard. To simplify the presentation of the logical properties, they define a number of layers on top of the basic logic to describe more complex properties.

The system ENCAP [111] by Taly et al. analyses programs written in the subset SES_{light} of JavaScript and reports accesses of security relevant resources which circumvent the trusted security APIs. The analysis is a context-insensitive, flow-insensitive points-to analysis. The function `eval` is handled conservatively by assuming the dynamically executed code aliases every variable with every possible value. ENCAP

derives its results by solving Datalog facts extracted from the program with conventional solvers. ENCAP could find all APIs accessing a critical resource to construct resource models for AmorJiSe and PhoneWrap. Otherwise, the results of ENCAP are orthogonal to the bounds provided here, since ENCAP only finds or certifies the absence of a resource access as opposed to inferring information about how often the resources are accessed.

RATA [77] is an abstract interpretation system, which aims to increase the performance of Just-in-Time compilers for JavaScript at runtime. RATA uses 3 different analysis methods: interval analysis to discover bounds on the values of numeric expressions, kind analysis to discover NaN values and fractional values and variation analysis to discover relations between different variables. The 3 methods complement each other to increase precision. The results are abstract values which contain information about the range of values possibly stored in the variables and whether they are compatible with Int32 values. This information is used to optimise the compilation of the code. The language covered is a core language named JavaScript$^{=}$. With respect to resource analysis, the results of RATA are especially interesting for the memory resource. The range of values for an expression determines the worst-case for their resource consumption in the memory and can therefore increase the precision of the requirement estimation.

2.3.3 Hybrid analysis

Some existing systems combine dynamic analysis with static analysis to improve the results.

Schäfer et al. [104] use an instrumented execution to track information about the determinacy of expressions. This system tracks the information during one trace generally enough, to return sound information about all traces. The results can be used to increase the precision of static analysis, for example, by replacing computed arguments of `eval` statements by their possible values. This method has been applied to different versions of the `jQuery` library and benchmarks of previous research on `eval` elimination. Newer versions of `jQuery` cannot be handled due to its complex use of event handlers.

Chugh et al. [22] approach the analysis of JavaScript code including dynamically loaded code. The system infers the behaviour of the available code statically and then describe properties the dynamic code has to fulfil to guarantee static properties of the

whole program. The properties of the dynamic code are derived automatically by a constraints based solution algorithm. The system then inserts dynamic checkers for the required properties into the code to enforce the static properties. The soundness of the results are proven formally for a subset of the JavaScript language. The full system is tested on the JavaScript code of the top 100 web pages.

The static part of the system Gatekeeper [47] uses points to analysis and dataflow constant propagation to prove the absence of certain attacks like global namespace polution and cross-site-scripting (XSS). These guarantees are derived by iteration of Datalog facts. This analysis is proven sound on a subset JavaScript_Safe of JavaScript, similar to the strict mode of EcmaScript 5. The `eval` function and other language constructs not covered by the subset are handled by runtime checks.

Wei et al. [117] propose a hybrid system which executes the code on a set of assumed comprehensible test cases using a modified version of the WebKit JavaScript engine. Based on the information obtained during these test executions, the static analysis then derives information about the taint propagation and aims to prevent leakage of personal data. The result are pairs of sources and sinks between which the critical data flows. For the dynamic features of JavaScript the system assumes that all possible values for the dymanic code are encountered during the tests.

Hackett and Guo [51] use static type inference to improve the benefit of Just-in-Time compilation for JavaScript. The focus of this work lies on the fact that every JavaScript expression has a polymorphic type since e.g. every object lookup might return `undefined` and integer arithmetic might overflow into a double value. They use dynamic enforcement to guarantee monomorphic types for the static part. A similar approach could be taken to handle the corner cases in the type system AmorJiSe.

2.4 Resource analysis

Most of the systems covered so far have qualitative properties as goal, like the absence of errors or threats. The analysis of the resource consumption of JavaScript code benefits from these results, but requires additional methods to capture the quantitative behaviour of the analysed JavaScript code. The following presents the two most evolved approaches to resource analysis.

2.4.0.1 Amortised types

The work initiated by Martin Hofmann in [54] introduces amortised types. The system AmorJiSe presented in this thesis takes the idea of amortised types and applies it to mobile resources in JavaScript.

In the amortised paradigm every data structure includes a certain number of reserved allocated memory units (only the resource memory is considered in the original work). The paper [54] introduces the type $\diamond$ that symbolised the allowance to allocate one additional cell in the memory. For example, the type $\diamond \to Int \to List(Int) \to List(Int)$ of the function `insert()` certifies that `insert()` requires one ticket, an integer value and a list of integers and returns a list. The provided resource unit has been used to insert the new value into the provided list.

The expressivity of the system was extended in [55] by replacing the unit $\diamond$ by annotations representing a list of units. Since the order of the tickets does not matter the annotation is represented as a numeric value, the list's length. Those annotations can be inserted into function types and data types. A number of tickets inside a data type can be thought of as a portion of the global freelist reserved for operations on this data structure. In recursive data structures, such a part of the freelist is associated with every element. A list of type $List(Int,2)$ has the allowance of allocating two additional memory cell per element in the list. This allows the list to be copied, with the result type $List(Int,1)$ for the original list and a freshly created list of the type $List(Int,0)$. The missing ticket in each element has been used to store the new copy. With this kind of type it is possible to reason about resource usage dependent on the size of the data structure.

Hofmann defines the typing rules for this system and presents an algorithm to infer these types automatically. The inference is achieved by first typing the expressions without resource annotations. Based on this type derivation the algorithm then gathers constraints on the resource annotations. The constraints form a linear programming problem (LPP) and can be solved for rational values (i.e. if resource units can be broken into fractions) or for integer values in special cases, e.g. if the coefficients in the LPP fulfil certain conditions.

Amortised types have been extended to handle non-destroying use of variables [12] and class-based programming languages [56]. An inference algorithm for the latter is presented in [57]. This is achieved by making the typing rules syntax-driven. Thus, the generation of the constraints is deterministic and can be solved as before. In this

implementation Hofmann assumes to have sufficient annotations from the user for the inference of the resource consumption. In particular the user needs to specify the resource relation were a variable is accessed more than once. For example for the consecutive function calls `f1(x);f2(x)` (both using `x`) the developer needs to declare how much of the potential of `x` will be available during the execution of f_1, how much will be available during f_2 and how much will be left with the variable `x` afterwards.

In related research, *linear types*, based on linear logic [43], are often use to type values which can only be used once, just like the allowance symbolised by the original $\diamond$ type can only be used once. However, advanced versions of amortised types allow a typed value to be read arbitrary often, as long as it is not used in resource consuming actions.

For example, consider the upload bandwidth as resource. Was the variable `x` in the code

Example 2.4.1.

```
1 var x = {value: "1.jpg", next: null};
2 upload_list(x);
3 show_list(x);
```

typed with a linear type, its type would be consumed in the function call `upload_list` and the following call `show_list` could not be typed. In amortised types, the call in line 2 merely consumes the annotation inside the type of `x` by 1, which limits the number of calls to the function `upload_list` but still allows resource independent calls like `show_list`.

Other linear type systems have been designed to only type a subset of the values with linear types and allow other values to be typed with non-linear types [116]. However, amortised type systems allow one value to be limited for some uses, but unlimited for others.

2.4.0.2 Cost relations

The *cost relation* approach [4], labels the cost of every subexpression with a variable and translates the program into a set of equations over those variables. The solution algorithm first transforms the equation system to eliminate indirect loops. The cost of direct loops is then estimated by over-approximating the worst-case for the number of loop iterations and the worst-case cost for one loop iteration. These values are obtained by an iterative procedure to obtain loop invariants and a monotonically decreasing

ranking function for the iterations. The product of those estimations produces a bound on the cost of the execution of the loop.

In the follow-up paper [5] this idea is refined to reason about the worst-case cost for each loop iteration separately. This results in tighter approximations. The algorithm does not claim to be complete in means of transforming the cost relations nor for finding the ranking functions and invariants. Experimental results show the analysis to be a good estimate for well known samples like merge-sort or the recursive solution for the Hanoi riddle.

Figure 2.11 Messaging plugin APIs

Plugin	Plugin name in JavaScript	JavaScript API Object	JavaScript API methods	Java API class	Java API methods	
org.apache.cordova.plugin.sms.Sms	cordova/plugin/Sms	window.SMS	send	Sms	send	github.com/aharris88/phonegap-sms-plugin
info.asankan.phonegap.smsplugin.SmsPlugin	cordova/plugin/SMSPlugin	smsplugin	send isSupported startReception stopReception	SmsPlugin	SEND_SMS HAS_SMS_POSSIBILITY RECEIVE_SMS STOP_RECEIVE_SMS	github.com/asanka-x/Phonegap-SMS
com.cordova.plugins.sms.Sms	cordova/plugin/sms	window.sms	send	Sms	send	github.com/cordova-sms/cordova-sms-plugin
com.dileepindia.cordova.sms.SMSPlugin	cordova/plugin/SMS	window.SMS	setOptions startWatch stopWatch enableIntercept sendSMS listSMS deleteSMS restoreSMS	SMS	setOptions startWatch stopWatch enableIntercept sendSMS listSMS deleteSMS restoreSMS	github.com/DILEEP-YADAV/ smsreader-dileepindia-cordova-plugin
com.rjfun.cordova.sms.SMSPlugin	cordova/plugin/SMS	window.SMS	setOptions startWatch stopWatch enableIntercept sendSMS listSMS deleteSMS restoreSMS	SMS	setOptions startWatch stopWatch enableIntercept sendSMS listSMS deleteSMS restoreSMS	github.com/floatinghotpot/cordova-plugin-sms/
org.apache.cordova.SMSComposer	-	window.plugins.smsComposer	showSMSComposer	SMSComposer	HasSMSPossibility showSMSComposer	github.com/GalCohen/SMSComposer-phonegap-plugin
tr.bel.mamak.sms_plagin [sic]	cordova/plugin/Sms	window.sms	send	Sms	send	github.com/hakkism/sms_REPO
com.jsmobile.plugins.sms.Sms	cordova/plugin/sms	window.sms	sendMessage	Sms	sendMessage	github.com/hazems/cordova-sms-plugin
info.ivanezko.phonegap.smsplugin.SmsPlugin	info.ivanezko.phonegap .smsplugin.smsplugin	window.plugins.sms	send	SmsPlugin	SendSMS	github.com/Ivanezko/sms-sender
org.apache.cordova.plugin.SmsPlugin		window.plugins.sms	send	SmsPlugin	SendSMS	github.com/javatechig/phonegap-sms-plugin
com.moga.plugins.smsplugin.SmsPlugin	-	window.plugins.smsPlugin	read send	SmsPlugin	ReadSMS SendSMS	github.com/jordan-carl/phonegap-plugins-1/ tree/master/Android/SMSPlugin
org.apache.cordova.plugin.SmsSendingPlugin	cordova/plugin/smssendingplugin	-	isSupported send	SmsSendingPlugin	HasSMSPossibility SendSMS	github.com/Pyo25/phonegap-sms-sending-plugin
net.practicaldeveloper.phonegap.plugins.SmsPlugin	cordova/plugin/sms	-	send	SmsPlugin	SendSMS	github.com/stormwild/SMSPlugin

Chapter 3

Interaction-dependent ticket-based policies

This chapter introduces interaction-dependent ticket-based policies (ITPs) and shows how to dynamically enforce them in JavaScript apps via API wrapping, a variation of inlined reference monitors. ITPs enforce a bound on the resource usage of an app during runtime by managing resource tickets. Each ticket grants a one-time resource access. Selected user interactions which are expected to trigger legitimate resource consumption generate additional tickets to "pay" for the expected resource access. Due to this mechanism, the resulting bounds depend on the user interaction with the app. Policy violations trigger the specified deny behaviour which can execute arbitrary JavaScript code to respond to the resource request and preserve as much functionality as possible without access to the resource.

Inlined reference monitors have been used before to implement policies for JavaScript web-apps [94] and Android apps [15]. Quantitative policies [33] and interaction-dependent policies [102] for mobile apps have been studied separately. This work, to the best of my knowledge, is the first to combine the different fields and to consider quantitative interaction-dependent policies for JavaScript apps. The result is the first system to enforce them in an unmodified execution environment.

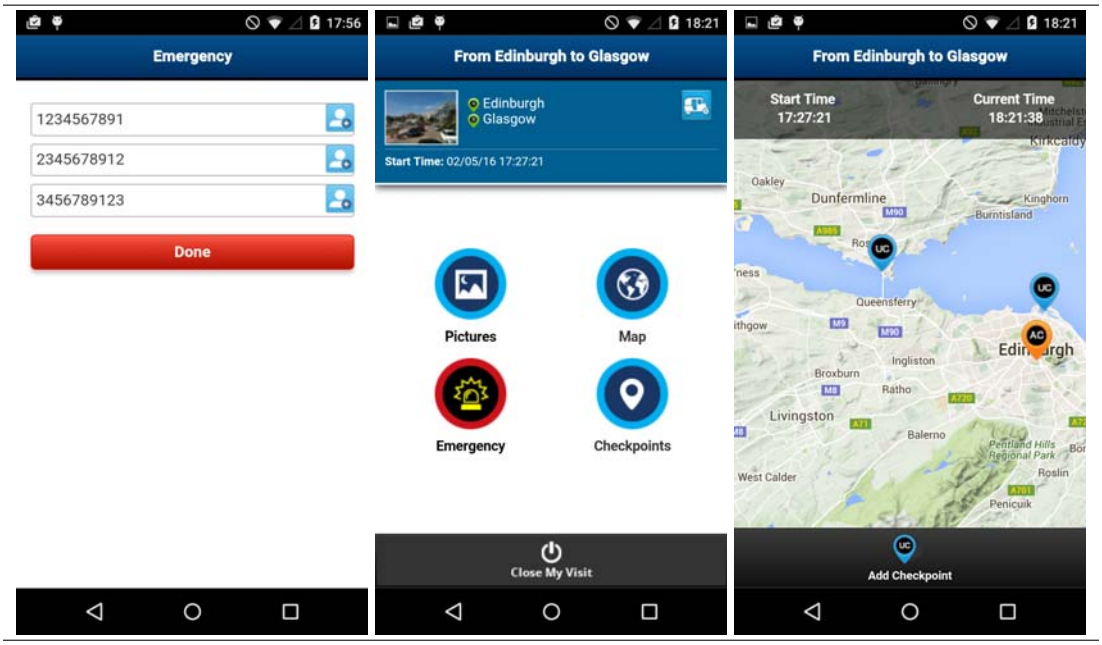
3.1 Motivating example

The PhoneGap app TrackMyVisit (myzealit.TMV.apk[1]) by MYZEAL IT Solutions shown in Figure 3.1 manages a list of journeys and advertises the following features:

[1] downloaded from the Google Play Store in Dec 2014, now discontinued

- While a journey is active the app logs the GPS position of the user.
- When the user presses the “Emergency” button, a message is sent to up to 3 specified contacts.
- With the button “Add Image” the app can take pictures and attach them to the journey log.

Figure 3.1 TrackMyVisit - UI

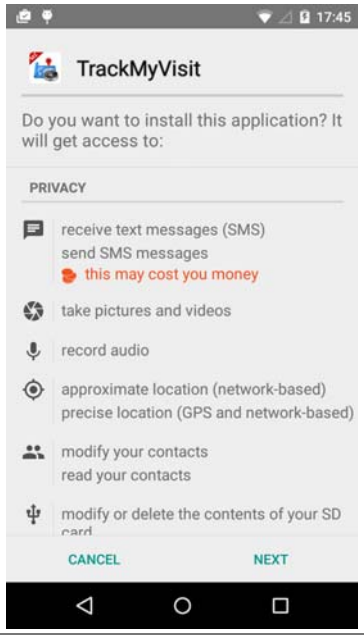


For this functionality the app requests, among others, the permissions to access the messaging service, the camera and the GPS location (Figure 3.2). The unrestricted access granted by these permissions enables the app to perform a number of attacks:

- The app could track the location of the user at all times.
- The app could take pictures in sensitive situations.
- The app could send private information or impersonate the user via messages and charge for premium messages.

More fine-grained control is required to allow the functionality but prevent these attacks. A true least-privilege policy for TrackMyVisit would grant access to the GPS sensor only while a journey is active, would allow exactly 1 camera access for each time the “Add Image” button is pressed and would permit messages only after the “Emergency” button has been pressed.

Figure 3.2 TrackMyVisit - Permissions



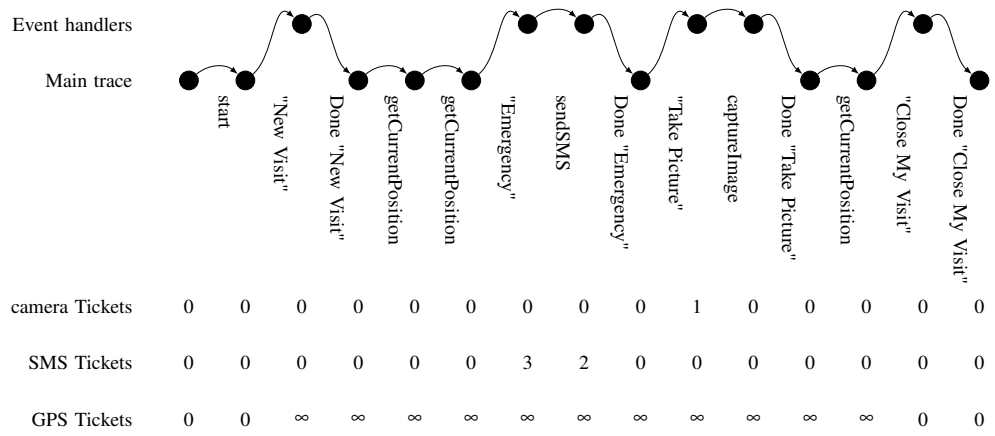
To achieve this, the *interaction-dependent ticket-based policies* (ITPs) manage a set of tickets which are granted to the app at launch or generated by specific user interactions. Each ticket allows a one-time access to the critical API. This way, the app is restricted to access the guarded resource exactly as often as needed to execute the functionality requested by the user. To enforce ITPs, the resource consuming APIs and the user interaction need to be monitored. The *wrapping* method monitors the APIs by overwriting the original API with an instrumented version which has sole access to the original API. Each time the application code calls one of the resource accessing functions, the wrapped API first evaluates and updates the policy and either calls the original API or executes the deny behaviour. The user interactions are monitored by listening for the user interaction events.

Figure 3.3 illustrates the effect of the policy described above. These example *resource traces* of the TrackMyVisit app record the button events and the API calls. Trace (a) conforms with the policy and shows how the button “Take Picture” generates 1 camera ticket to be used by the API function `captureImage`. The button “Emergency” generates 3 SMS tickets, one of which is consumed by the `sendSMS` function. The remaining 2 tickets are cancelled when the event handler for the “Emergency” event is finished. Finally the buttons “New Visit” and “Close My Visit” enable or disable unconditional access to the GPS sensor by granting ∞ many tickets. Trace (b) does not conform with the policy since it calls the API `getCurrentLocation` before the button “New Visit” started a journey and after “Close My Visit” was pressed. In

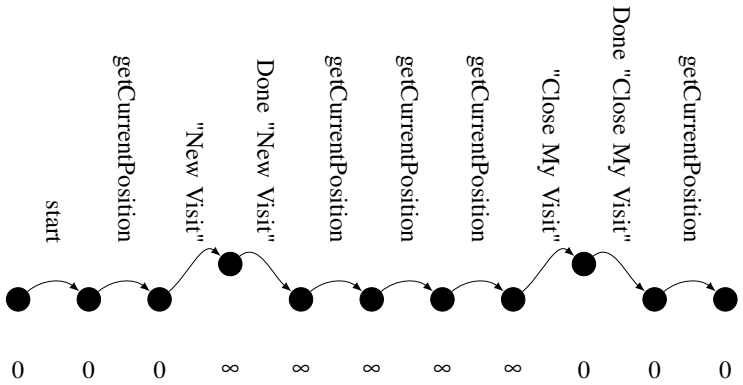
both cases the app does not have a GPS ticket to pay for the access. As a consequence, the enforced trace (c) replaces the violating requests by the deny action. It does not restrict the legitimate resource access while the journey is active.

Figure 3.3 Example traces of a ticket-based permissions model

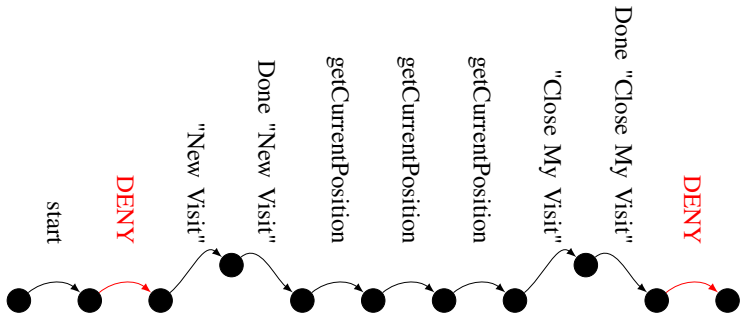
(a) Conforming trace



(b) Violating trace



(c) Enforced trace



3.2 Policy model

This section describes the formal model of the interaction-dependent ticket-based policies (ITPs). Here, each ITP guards one abstract resource. To guard multiple resources, multiple independent ITPs can be defined or multiple resources can be combined into one abstract resource. For example, the abstract "notification" resource contains pop-up dialogs, vibration and audio signals. Each notification ticket can be used to activate either of the guarded features.

3.2.1 Definition

The following describes the formal concept of ITPs. The policy reacts to API calls f and user events $e \in \mathit{UIEvents} = \{\texttt{click(button)}, \texttt{mousedown(x,y)}, ...\}$. The special event $\texttt{start} \in \mathit{UIEvents}$ indicates the launch of the app.

Definition 3.2.1.

- Let a *resource model* be a function

$$RM : API \to \{0,1\}$$

  with $RM(f) = 1$ if the function f accesses the guarded resource and $RM(f) = 0$ otherwise. Call APIs f with $RM(f) = 1$ *critical API*.

- Let an *interaction policy ip* be a pair of functions

$$ip_l, ip_g : \mathit{UIEvents} \to \mathbb{Q}_\infty^{\geq 0}$$

  defining the amount of *local* and *global* tickets generated for each event.

- Define a *full policy* as the triple $pol = (RM, ip, deny)$ with the deny behaviour *deny*.

The interaction policy $ip = (ip_l, ip_g)$ describes two different kinds of tickets. Global tickets ip_g can be used at any time after generation, whereas local tickets ip_l are generated for a specific event. They can only be used within the event handlers for the event they were generated for and unused local tickets are cancelled after the event has been handled. Both values ip_l and ip_g are taken from $\mathbb{Q}_\infty^{\geq 0} = \{x \in \mathbb{Q} | x \geq 0\} \cup \{\infty\}$. The value ∞ is included to describe unrestricted access equivalently to an Android permission. With fractional value in $\mathbb{Q}$, the system can describe fractions of a ticket. This

way the policy can require multiple user interactions per resource access and weight different user interaction differently. The values are taken from $\mathbb{Q}$ rather than $\mathbb{R}$ to capture the nature of the implementation better, which cannot represent arbitrary real numbers.

The function *deny* describes the behaviour which is executed in case the app requests resource access without sufficient tickets. The *deny* behaviour may, for example, terminate the application, ignore the resource request, return dummy values, grant access to the resource based on additional conditions (e.g. a user confirmation) and even manipulate the number of tickets granted to the app.

Definition 3.2.2. Consider an app P.

1. Consider the following *resource events* r:

   (a) $API(f)$: call to the API f

   (b) $E(e)$: the user triggers the event $e \in UIEvents$

   (c) $Done(e)$: all handlers for the event $e \in UIEvents$ are completed

   Enforced traces can also include the additional resource event

   (d) *deny*: the deny behaviour is executed

2. Define a *trace* t as the possibly infinite sequence $r_1, r_2, ...$ of resource events occurring during an execution of an app P.
   Write $t \cdot t'$ for the concatenation of two traces t and t' and $P \rightarrow_t^*$ to assert that the app P produces the trace t.

3. For each trace $t = (r_1, r_2, ...)$ define the *resource count* c_{res} as:

$$c_{res}(r_1, r_2, ...) = \sum_{r_i = API(f)} RM(f)$$

4. For a trace $t = t' \cdot (E(e), ..., Done(e))$, let the *event trace* $t|_e$ be the shortest suffix $E(e), ..., Done(e)$ of t.

5. Provided a policy $pol = (RM, ip, deny)$, define the *ticket count* c_{tic} of a finite trace $r_1, ..., r_k$ recursively as

$$\begin{aligned}
c_{tic}(\varepsilon) &= 0 \\
c_{tic}(t' \cdot API(f)) &= c_{tic}(t') - RM(f) \\
c_{tic}(t' \cdot E(e)) &= c_{tic}(t') + ip_l(e) + ip_g(e) \\
c_{tic}(t' \cdot Done(e)) &= c_{tic}(t') + \min(0, c_{res}(t|_e) - ip_l(e))
\end{aligned}$$

with $t = t' \cdot Done(e)$ in the last case. The number of tickets $c_{tic}(t' \cdot deny)$ in the deny case can be freely set by the deny behaviour.

Since $RM(f) \geq 0$, the infinite sum $\sum_{r_i = API(f)} RM(f)$ for infinite traces is well behaved: it is either finite if $\exists i > 0 \forall r_j = API(f_j)$ with $j \geq i : RM(f_j) = 0$ or ∞ otherwise. The same holds true for the infinite sums in the following sections.

Intuitively, c_{res} tracks the number of used resources while c_{tic} tracks the number of available tickets. Each API call subtracts the correct amount of tickets while relevant events add new tickets. The term $\min(0, c_{res}(t|_e) - ip_l(e))$ subtracts unused local tickets at the end of an event scope.

JavaScript's "run-to-completion" model guarantees that in any valid trace each event is completely handled before the next event is considered. Therefore, the trace $t|_e$ is well defined and, except for the leading $E(e)$ and concluding $Done(e)$, contains only resource events $API(f)$ and potentially *deny* events. Note that, since $Done(\texttt{start})$ does not occur in any trace, the interaction policy $ip(\texttt{start}) = (n, m)$ is equivalent to $ip(\texttt{start}) = (0, n + m)$.

Definition 3.2.3.

1. A trace $t = r_1, r_2, \ldots$ *conforms* with the policy *pol*, written $pol \vdash t$, if there is no i such that $c_{tic}(r_1, \ldots, r_i) < 0$.

2. An app P *conforms* with the policy *pol*, written $pol \vdash P$, if for every trace t with $P \rightarrow_t^*$ it holds $pol \vdash t$.

In general, a deny behaviour can freely manipulate the available tickets by setting the value of $c_{tic}(t \cdot deny)$ and execute the original critical APIs. Call a deny behaviour *weak*, if it does not affect c_{tic} or c_{res}.

Definition 3.2.4. Let a deny behaviour *deny* be called *weak*, if

1. it does not affect the number of tickets: $c_{tic}(t' \cdot deny) = c_{tic}(t')$

2. it does not call critical APIs f

Equivalently, a policy $pol = (RM, ip, deny)$ is called *weak* if *deny* is weak.

3.2.2 Enforcement

The ITPs are enforced by wrapping all resource consuming APIs and relevant events. The wrapped functions encapsulate the original API and execute it according to the policy or call the deny behaviour otherwise.

Definition 3.2.5. A *policy state* $s = (c_l, c_g) \in \mathbb{R}_{\infty}^{\geq 0}$ describes the number of available local tickets c_l and global tickets c_g. The policy enforcement assumes that the policy state cannot be accessed by the original app P.

A policy state transition along the finite trace t is written as $(c_l, c_g) \rightarrow_t (c_l', c_g')$.

This definition allows each of the values c_l, c_g also to express partial tickets in $\mathbb{R}$. This way policies can generate partial tickets for each user interaction, weight different kinds of user interaction differently and require multiple interaction for each resource access.

Given the policy state, wrapping is defined as:

Definition 3.2.6.

1. For the API f, define the wrapped API $wrap_{pol}(f)$ as the following:

```
1 if (c_l ≥ 1)
2     c_l = c_l - 1;
3     call f;
4 else if (c_g + c_l ≥ 1) //0 ≤ c_l < 1
5     c_l = 0
6     c_g = c_g - (1 - c_l);
7     call f;
8 else
9     call deny;
```

2. Define the wrapped event $wrap_{pol}(e)$ as the event e' which first updates the policy state (c_l, c_g) according to ip, executes the original handlers for event e and finally sets $c_l = 0$.

3. Define the wrapped app $wrap_{pol}(P)$ as the app executing P in an environment where the critical API f has been wrapped as $wrap_{pol}(f)$ and every event e is handled as $wrap_{pol}(e)$.

The wrapping is assumed to be *complete*:

- Every call to guarded APIs with $RM(f) \neq 0$ is replaced by its wrapped API $wrap_{pol}(f)$.

- The app P cannot call a guarded function f otherwise.

Remark 3.2.7. In the wrapper for an API $wrap_{pol}(f)$ local tickets are consumed preferred, since their validity period is definitely shorten than the global tickets. This way, the app has the chance to use the tickets to their full potential. Even if only a partial local ticket $0 < c_l < 1$ remains, this partial ticket is consumed first, assuming the remaining part to combine a full ticket can be accounted for by the global ticket counter. For this purpose line 4 checks, whether the sum $c_g + c_l \geq 1$ instead of $c_g \geq 1$. If this check is true, line 5 consumes the partial local tickets and line 6 reduces the global ticket counter by the remaining part.

3.2.3 Properties

Lemma 3.2.8. Given a weak policy *pol* and a conforming trace $pol \vdash t$, the resource consumption of $t = r_1, r_2, ...$ is bounded by

$$c_{res}(r_1, r_2, ...) \leq \sum_{r_i = E(e)} \big(ip_l(e) + ip_g(e)\big)$$

Proof. It holds $c_{res}(\varepsilon) = 0 = c_{tic}(\varepsilon)$. Every time c_{res} increases (for some $r = API(f)$) c_{tic} decreases. Since *deny* is weak, only the events ($r = E(e)$) increase c_{tic}, in total by

$$\sum_{r_i = E(e)} \big(ip_l(e) + ip_g(e)\big).$$

The claim follows from $c_{tic}(t) \geq 0$ for the conforming trace t. □

Lemma 3.2.9. For a finite trace t, the policy state $(0,0) \rightarrow_t (c_l, c_g)$ of a weak policy tracks the available tickets:

$$c_{tic}(t) = c_l + c_g$$

Proof. This can be proven by induction on the length of the trace t. The only interesting case is $t = (t' \cdot Done(e))$:

- Assume $(0,0) \rightarrow_{t'} (c'_l, c'_g)$ with $c_{tic}(t') = c'_l + c'_g$.
- Since $t = (t' \cdot Done(e))$ it holds $c_l = 0$, $c_g = c'_g$.

Split $t = t'' \cdot t|_e$. Due to JavaScript's "run-to-completion" model, for every event $E(e)$ in t'' also the accompanying $Done(e)$ event occurs in t''. Therefore, $(0,0) \rightarrow_{t''} (0, c''_g) \rightarrow_{E(e)} (ip_l(e), c''_g + ip_g(e)) \rightarrow_{...} (c_l, c_g)$. Remember, that $t|_e$ only contains API calls. Due to the definition of $wrap_{pol}(f)$ it holds that $c_l + c_g = ip_l(e) + c''_g + ip_g(e) - c_{res}(t|_e)$

If $c_{res}(t|_e) \geq ip_l(e)$, then $c'_l = c_l = 0$ and

$$\begin{aligned} &min(0, c_{res}(t|_e) - ip_l(e)) = 0 \\ \Rightarrow &c_{tic}(t) = c_{tic}(t') = c'_l + c'_g = c_l + c_g. \end{aligned}$$

Otherwise, if $c_{res}(t|_e) < ip_l(e)$, then $c'_l = ip_l(e) - c_{res}(t|_e)$ and $min(0, c_{res}(t|_e) - ip_l(e)) = -(ip_l(e) - c_{res}(t|_e))$. Therefore,

$$\begin{aligned} c_{tic}(t) =& c_{tic}(t') - (ip_l(e) - c_{res}(t|_e)) \\ =& c'_g + c'_l - (ip_l(e) - c_{res}(t|_e)) \\ =& c'_g = c_g = c_g + c_l. \end{aligned}$$

□

Lemma 3.2.10. A wrapped app conforms with its policy:

$$wrap_{pol}(P) \rightarrow_t^* \Rightarrow pol \vdash t$$

Proof. Assume the claim is not true, then choose a finite prefix $r_1, ..., r_i$ of t with $c_{tic}(r_1, ..., r_i) \geq 0$ and $c_{tic}(r_1, ..., r_{i+1}) < 0$. Since r_{i+1} decreases c_{tic}, it must hold $r_{i+1} = API(f)$. By assumption all critical API calls have been replaced by $wrap_{pol}(f)$ which only calls the original f if $c_{tic}(r_1, ..., r_i) = c_l + c_g \geq 1$. Due to the definition of c_{tic} that means $c_{tic}(r_1, ..., r_{i+1}) \geq 1 - RM(f) = 0$ which is a contradiction. □

For weak policies Lemma 3.2.8 and 3.2.10 provide an interaction-dependent bound limiting all possible traces of the injected app. For general policies this bound has to be adjusted to reflect the deny behaviour. Firstly, *deny* might access the original critical API directly for which c_{res} does not account. Secondly, *deny* might generate additional tickets allowing the application code access to the critical API through the wrapped API.

Corollary 3.2.11. *A wrapped app $wrap_{pol}(P)$ accesses the critical APIs less than the bound provided in Lemma 3.2.8 plus the number of generated tickets and critical API calls in the deny behaviour.*

3.3 Captured policies

ITPs cover a wide range of common policies including the following ones:

- **Deny all**: Without the permission for a resource, Android terminates every request to this resource with a security violation. An ITP emulates this behaviour by granting 0 tickets to the resource. In contrast to a rejected permission, ITPs can apply more versatile deny behaviours, which preserve the remaining functionalities of the app.

- **confirm each time**: Some security policies ask the user to confirm every resource access. For example, Google Chrome recommends the setting "Ask when a site wants to track your location" as policy for the Google location service. An ITP implements such a policy by asking the user as part of the deny behaviour. Since the deny function has access to the policy state, it can call the original function directly when the user confirms access.

- **confirm on first use**: The iOS system asks for the permission when a resource is accessed for the *first time* and then grants the permission until the app is closed. This way, in comparison to the Android permissions granted during installation, the user can make a more informed decision. An ITP can describe these policies by granting ∞ many global tickets in the deny behaviour after the user confirms access.

- **not before interaction**: Some resources are especially critical when they are accessed in the background. An ITP can grant ∞ many global tickets for each interaction. This grants access only after the first interaction.

- **only in interactions**: Some resources should never be accessed unless explicitly requested by the user. If the details of the accessing user interaction cannot be specified in advance, ITPs can generate ∞ many *local* tickets for each interaction, granting access only in event handlers.

3.4 Discussion

This chapter presented the concept of interaction-dependent ticket-based policies enforced by wrapping. It was shown that such policies enforce a bound on the resource usage dependent on the number and kind of the UI events. With these the resource behaviour of mobile apps can be described precisely enough to allow the functionality, while blocking potentially malicious behaviour.

From the resource classification given in Section 2.2.1 ITPs can guard the resources described as *count*, since each resource access is counted as 1 consumed unit and they cannot be released anymore. By relaxing the resource model to $RM : API \to \mathbb{Q}$, ITPs could also cover *cumulative*, *non-blocking unique* and *acquire-release* resources. Here, APIs f with $RM(f) < 0$ symbolise functions releasing resources. For *blocking unique* resources the policy needs to be further extended with a maximum amount of tickets the state can hold at any one time. In reality, the count category covers the majority of API activated resources and for this reason the policy language presented here is kept simple, sacrificing the expressivity. Further research into different resources might make it necessary to implement those extensions.

ITPs can guard almost all PhoneGap resource kinds discussed in Section 2.2.2. For resources like contacts, which require to create an instance first and then call the APIs as methods of the instance, ITPs can guard the method in the prototype of the plugin and effectively wrap all instances. Resources accessed based on events like `lowbattery`, `chargerconnected`, `online` and `offline` cannot be wrapped, since the policy model does not track the listeners to those events. Furthermore, for resources accessed by callback-functions like `watchAcceleration`, `watchHeading` and `watchPosition` ITPs can only restrict how many callbacks are registered, not how often they are called. Policies restricting the number of callbacks can be enforced by wrapping , but require a more complex wrapper altering the parameters and return values of the wrapped function.

The deny behaviours of the ITPs make it possible to define various existing policies as ITP. This feature makes the PhoneWrap policies equivalent to edit automata [75] with its ticket counter as a countable infinite state set. However, it is worth noting that such edit automata can change the behaviour of the app arbitrarily, if the continuation of the trace depends on the outcome of the replaced API call.

This theoretical model of the wrapping has a few assumptions. Most critical it assumes, that the application code cannot access the wrapped assets (function handler and policy state). The implementation PhoneWrap, in the following chapter, uses the wrapping approach presented in [94]. This work already had to be corrected in [82], because these assumptions were not provided properly. The main issue was to store an unmodified copy of all built-in functions used inside the wrapper. Otherwise attacking code could overwrite the built-in functions and trick the wrapper to execute malicious code inside the wrapper due to JavaScript's dynamic scope. However, verifying or proving the separation of the wrapper from the application code is a field of research

in itself which could for example include pointer analysis.

Apart from wrapping, there are other methods which achieve a similar result: *Rewriting* modifies the JavaScript code to alter its behaviour. This approach has been used in web-based applications in previous work [78, 16]. The rewriting method also needs to wrap all code inserted or modified dynamically at runtime which is a often used feature in JavaScript. In comparison, the wrapping proposed here modifies the dynamic execution environment by wrapping the original API such that all code inserted after the wrapping procedure is already guarded by the policy. This approach has a lower runtime over-head and provides less attack surface for a potential policy bypass.

ITPs could also be enforced on a lower level of execution: PhoneGap apps execute their JavaScript code on top of the framework's Java code and the operating system. Instrumenting the framework level would simplify the specification of the critical APIs, but does not have access to the user interaction, since interaction is handled in the HTML / JavaScript layer. Additionally, modifying the operating system would not be as self-contained as the directly wrapping approach inside the app package.

Previous work [84, 27, 93, 46, 113] modified lower-level systems like the browser-view or the operating system. However, modifying these systems requires root access to the phone and needs to be re-applied for every update of the system.

ITPs only regulate how often the guarded resource is accessed, not how it is used after access. After an app has obtained private information it can share this information freely. To protect the user against private information leakage, orthogonal methods like flow-analysis are available [93, 27, 46].

Chapter 4

The system PhoneWrap

This chapter presents the system PhoneWrap which injects ITPs into real-world PhoneGap apps. To the best of my knowledge, PhoneWrap is the first system to enforce quantitative interaction-dependent policies in unmodified execution environments.[1]

The system has been developed with the following core aims:

Ease of Use Users with a basic understanding of the user interface and resource behaviour of the guarded app and the concept of ticket-based policies are able to create PhoneWrap policies. Specifically, PhoneWrap does not assume knowledge of the app's source code or the enforcement method.

Stand-alone PhoneWrap's enforcement is contained within the guarded app and executes in an unmodified environment. Modifications of the execution environment require root access to the phone which undermines important security principles and many users are not able or willing to make this modification.

Real-world Real-world apps consist of the JavaScript code but also include the framework configuration and plugins. PhoneWrap automatically extracts all necessary information from a standard Android app package and after policy injection repacks the app into a package directly installable on the target phone.

The resulting tool is a proof-of-concept implementation and does not claim to be resilient against all possible attacks, e.g. through global namespace pollution as shown in [82].

Unless explicitly stated, this chapter discusses the injection of a policy enforcing bounds on one resource, for example messaging. If multiple resources have to be

[1] The results of this chapter with some additions from the previous chapter have been published at the International Workshop on Innovations in Mobile Privacy and Security (IMPS, 2016) [40].

guarded, PhoneWrap can be used to insert a separate policy for each resource. This results in separate policy states and different tickets for each resource.

4.1 Terminology

The system PhoneWrap injects a *wrapper script* into a PhoneGap *app* to enforce a bound on the usage of the *guarded resource*. The app behaviour consists of *expected functionality* and *potentially unwanted* resource access and is implemented in JavaScript with access to Java functionality implemented in *native code*. The injected wrapper script contains the *policy* describing how often and in which *context* the *application code* may access the resource to execute the functionality.

Within the wrapping script PhoneWrap defines the *wrapper function* and uses it to dynamically instrument all *resource consuming* or *critical APIs*. The bounds enforced by the policy are described in terms of *tickets* which each allow one-time access to the guarded resource. A *global* ticket can be used at any point during runtime while *local* tickets can only be used within the scope of the event by which the ticket was generated.

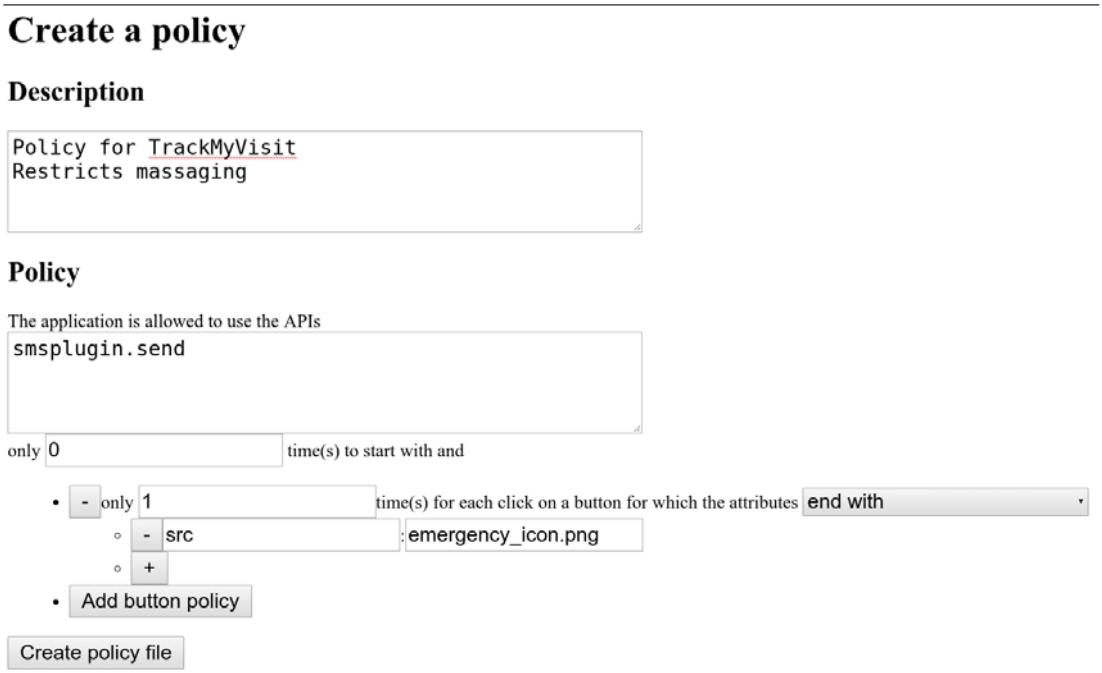
The *PhoneWrap scripts* execute the different steps needed to inject the wrapping script into the *app package*.

4.2 Policy specification

The PhoneWrap policies are specified as a JSON / JavaScript object. Each field of the object represents a policy parameter. A detailed reference of all policy parameters accepted by PhoneWrap can be found in Appendix B.

PhoneGap app developers are already familiar with JSON / JavaScript and existing infrastructure and tools (such as editors, code highlighting and structured views) can be used to transfer policies over the Internet or visualise and modify the policy content. Policy authors not familiar with JavaScript can specify policies using PhoneWrap's HTML form shown in Figure 4.1, which embeds the policy parameters into template sentences.

Figure 4.1 Create a policy



4.2.1 Policy parts

Essentially, PhoneWrap policies consist of the 3 parts specified in the ITPs (see Definition 3.2.1):

Guarded resource

The *guarded resources* are described by their accessing APIs in three different categories: the policy parameter `guard` describes the critical JavaScript APIs, `guard_exec` describes the critical Java APIs and `guard_require` describes the critical plugins. PhoneWrap needs to guard these three different types of APIs in different ways as discussed in detail in Section 4.3.1.

Interaction policy

The *interaction policy* in PhoneWrap is implemented as a finite set BP of button policies. Each $bp \in BP$ is specified via the triple $(cond, mperms, local)$. At runtime each occurring UI event specifies the *event target* as the UI element the event was triggered by. When an event occurs, the parameter *cond* of each button policy *bp* is evaluated against the event target and, if it matches, *mperms* tickets are created. If the flag *local* of *bp* is true, the generated tickets are marked as local to this event.

A set BP describes a formal interaction policy ip in the following way:

$$\begin{aligned} ip_l(e) &= \sum_{\substack{bp \in BP \\ bp.local \\ bp.cond(e)}} bp.mperms \\ ip_l(\texttt{start}) &= 0 \\ ip_g(e) &= \sum_{\substack{bp \in BP \\ \neg bp.local \\ bp.cond(e)}} bp.mperms \\ ip_g(\texttt{start}) &= m_0 \end{aligned}$$

where m_0 is the number of tickets granted by this PhoneWrap policy at launch. Given $BP = \{bp_1, ..., bp_k\}$ and a trace $t = r_1, r_2, ...$ (see Definition 3.2.2), the security property of the ticket-based policy guarantees the bound

$$c_{res}(t) \leq m_0 + \sum_{i=1..k} (c_i(t) \times bp_i.mperms)$$

where

$$c_i(t) = |\{e \in t \mid bp_i.cond(e)\}|$$

is the number of events in t matching the conditions of policy bp_i. This bound guarantees that the number of resources units actually consumed by the trace t, is limited by the appropriate amount per button click. This bound does not factor in the cancelled local tickets, but instead assumes they are all used in their respective validity scope.

In the policy specification, the parameter `buttons` specifies the list BP of button policies. Each bp is defined as a nested JavaScript object with the field `cond`, `mperms` and `local`. The condition `cond` itself is an object. Each field of the `cond` object describes a property the event target has to fulfil for this button policy to apply.

Deny behaviour

The `deny` behaviour in PhoneWrap is defined as a JavaScript function with access to the policy state. This enables many expressive responses fine-tuned to the guarded resource, for example, terminating the app, ignoring the resource request, returning dummy values or interactive with user inquiry.

Policy state

During runtime PhoneWrap implements the *policy state* (c_l, c_g) of the ITPs via two counters `mperms_local` and `mperms_global`. The initial value of `mperms_global` is set by the policy parameter `mperms`.

4.2.2 Example policy

An example policy for the "TrackMyVisit" app is given in Figure 4.2. This policy guards the APIs to send messages specified via the parameters `guard`, `guard_require` and `guard_exec`. At app launch 1 ticket to access either of these APIs is granted.

The interaction policy `button` in this example only contains one button policy. This button policy generates 3 (`mperms`) local tickets for each click on an element with the "src" property ending in `images/emergency_icon2.png` (`cond`). The parameter `match` of the button policy determines how the values in `cond` are compared to the properties of the event target. Its value "end" in this policy describes that any button with an icon path ending in "images/emergency_icon2.png" activates this button policy. This accounts for the prefix of the absolute icon path during runtime. Possible values for the parameter `cond` include "exact", "different", "ends", "begins" and "contains" which each compare the properties of the occurring event with the property values specified in the policy in the self-explanatory way. More experienced policy authors can also use the matching method "regex" to specify the properties of the policy-activated UI elements as a JavaScript regular expression. The `deny` behaviour in this policy simply ignores the resource access and displays the message "Policy: Denied" instead.

Figure 4.2 Example policy

```
policy = {
  mperms : 1,
  buttons : [
    {
      cond: {
        src:"images/emergency_icon2.png"
      },
      mperms:3,
      match:"ends",
      local: true
    }
  ],
  guard: ["smsplugin.send"],
  guard_exec: ["SmsPlugin.SEND_SMS"],
  guard_require: ["cordova/plugin/smssendingplugin.send"],
  deny: function(){alert("Policy: Denied")}
}
```

4.2.3 Extensions

To capture the resource behaviour of real-world apps better, PhoneWrap offers a few features in addition to the core features of an ITP. The policy state has the switches

`allowAll` and `blockAll` which enable or disable all access regardless of the available tickets. The corresponding ITP would set $c_{tic} = \infty$ or $c_{tic} = -\infty$. The starting state of these switches can be specified in the `allowAll` and `blockAll` parameters in the policy and every button policy can assign a new value to those parameters. A PhoneWrap policy can also flag a button policy as `checkbox` or `confirm` which changes the way the policy grants tickets for events on those elements. Section 4.3.2.1 discusses in which situations these policy behaviours are needed.

4.3 Policy enforcement

Inspired by the method proposed in [94], PhoneWrap provides the *wrapper script* which contains the policy and the code to wrap the necessary functions. PhoneWrap inserts this script into the header of the main HTML file and wraps the relevant assets at multiple stages during the life-cycle of the app. The overall layout of the script is as follows:

```
1 function wrapper () { //all locals inside this body are protected
2     // Define policy object
3     // wrap API functions
4     // wrap input elements
5     // wrap auxiliary functions
6 };
7 wrapper();
```

The whole policy is implemented inside a function body. When this function is called, JavaScript creates a new scope object for the function body in which PhoneWrap stores the policy state and the original resource consuming APIs as local variables. The JavaScript scope ensures that local variables cannot be accessed from outside the function body and consequently isolates the policy from the application code. The wrapper script all together contains ~500 LOC which does not influence the package size significantly.

The last step, wrapping the auxiliary functions, includes listening for freshly inserted JavaScript scripts into the DOM tree of the app. New scripts might contain new APIs which have to be wrapped, before the app uses them. For this reason, PhoneWrap wraps all available critical API functions each time a new script is inserted. This ensures all critical APIs are wrapped at all times. Since it is not trivial to check whether a function is already wrapped, PhoneWrap wraps all available function each time the

wrapping is called. Should a function be wrapped multiple times, PhoneWrap takes care, that only one ticket is consumed, independent of the number of wrapping layers.

4.3.1 Wrapping API functions

Inside the wrapper script PhoneWrap copies all critical APIs into local variables and then overwrites the public references by a wrapped version. This includs the APIs specified by the policy and further administrative functions to guard the integrity of the policy. The function `exec_guard` (Figure 4.3), included in the wrapping script, wraps a function appropriately corresponding to $wrap_{pol}$ of the ITPs. As input it expects a function call consisting of the name of the function and the actual parameter values. When called, it evaluates the policy state and either executes the deny behaviour or the original function call. During the wrapping procedure the references to all specified APIs are overwritten with their wrapped versions in the JavaScript execution environment.

Inside the wrapper, the policy state is checked and modified in the following order:

1. if `blockAll` is true → deny access
2. if `allowAll` is true → allow access
3. if at least one local ticket is available → allow access and subtract one local ticket
4. if at least one ticket is available (global + local) → allow access, delete all fractional local tickets and global tickets to add up to 1 full ticket
5. otherwise → deny

This policy enforcement is applied to 3 different kinds of APIs:

Java APIs As discussed in Section 2.1.4, PhoneGap plugins are implemented as Java classes. Via the PhoneGap bridge function `exec` JavaScript code can call the Java API directly. For example,

```
cordova.exec(..., "SmsPlugin","SEND_SMS",parameters)
```

calls the method `SEND_SMS` of the plugin class `SmsPlugin` with the provided `parameters`. PhoneWrap instruments the function `exec` to appropriately enforce the policy should `exec` be called with a Java API as specified in the parameter `guard_exec` of the policy.

Figure 4.3 Wrapping function

```
1 var exec_guarded = function(f,f_this,f_arguments) {
2   var allowed_global = (policy.mperms_global + policy.
        mperms_local > 0);
3   var allowed_local  = (policy.mperms_local  > 0);
4   var perms_pre;
5   if (!policy.blockAll) {
6     if (policy.allowAll) {
7       var back = f.apply(f_this,f_arguments);
8       return back;
9     }
10    if (allowed_local) {
11      perms_pre = policy.mperms_local;
12      var back = f.apply(f_this,f_arguments);
13      policy.mperms_local = perms_pre -1; //only ever decrease by
            1, even if wrapped by this policy multiple times
14      return back;
15    }
16    if (allowed_global) {
17      perms_pre = policy.mperms_global+policy.mperms_local;
18      var back = f.apply(f_this,f_arguments);
19      policy.mperms_local = 0;
20      policy.mperms_global = perms_pre -1; //only ever decrease
            by 1, even if wrapped by this policy multiple times
21      return back;
22    }
23  }
24  return policy.deny(f_this,f_arguments);
25 }
```

JavaScript APIs Most plugins provide a JavaScript interface internally calls the Java API using the `exec` function. For example, `smsplugin.send` calls the Java API above. Recent versions of PhoneGap initialise these JavaScript interfaces during the start-up phase. Since this initialisation is performed before the wrapping of `exec`, they need to be wrapped individually.

PhoneWrap instruments all JavaScript functions specified in the `guard` parameter of the policy.

Plugins A fresh copy of the JavaScript API of a plugin can be requested using the `require` function. For example, JavaScript code obtains the API for the SMS plugin by calling `require("cordova/plugin/smssendingplugin")`. This is most commonly used in early versions of the PhoneGap framework which do not initialise the JavaScript APIs automatically. PhoneWrap instruments the `require` function to automatically wrap the critical functions specified in the policy parameter `guard_require` before the API is forwarded to the application code.

JavaScript's inheritance is based on a prototype chain, which defines the parent object for each individual object. Before any function is wrapped, PhoneWrap fol-

lows the prototype chain and wraps the critical function in the most general object to guarantee wrapping everywhere in the relevant object hierarchy.

When the wrapping script is executed all available APIs are wrapped immediately. After the initial wrapping, PhoneWrap listens for further critical APIs to become available and (re-)wraps all APIs defined in the policy accordingly:

- PhoneWrap listens for the event `deviceready` sent by PhoneGap once all libraries are initialised and wraps all available APIs. To receive this event in older versions of the framework, PhoneWrap re-registers the handler after the PhoneGap library has been loaded.

- PhoneWrap registers as an HTML MutationObserver to discover additional scripts loaded into the app. Every time a new script is loaded into the DOM tree, PhoneWrap executes the wrapping to cover all APIs defined in this script. This includes scripts included in the original DOM tree and dynamically inserted script.

Since all critical APIs are wrapped at multiple points of the app's execution and PhoneWrap wraps the APIs on different layers (Java API/JavaScript API/Plugin), the APIs might be wrapped with multiple layers of wrapper. Depending on the implementation of the particular plugin, the API function might be wrapped once through the Java API, and once (either through the plugin or JavaScript API) per included script in the app's DOM tree. A usual app contains it's main script, the cordova library file and several auxiliary or advertisements script. PhoneWrap, however, only consumes one ticket per call, independent of the number of wrappers.

4.3.2 Interaction dependencies

The novel advantage of PhoneWrap are the interaction-dependencies. PhoneWrap listens for user events via the standard HTML event queue and generates tickets for each event according to the policy. In the HTML event standard each event is first sent to all of the parents of the target node, starting at the root node of the DOM tree. PhoneWrap utilises this and attaches an event handler to the root to receive all events.

For each event, the PhoneWrap event handler merges the effects of all matching button policies before the accumulated effect is applied to the policy state. The local and global tickets generated in each matching button policy are simply summed up. The switches `blockAll` and `allowAll` are handled in a conservative way: If no

matching button policy assigns a new value to `allowAll` then the values in the policy stay unchanged. If the newly assigned values are either all `true` or all `false`, then this new values is assigned to the policy state. If some matching button policies assign `true` and other assign `false`, then `false` is accepted as new value in the policy state, as this is the fail-safe behaviour. The parameter `blockAll` has the dual behaviour and accepts `true` in case `true` and `false` is assigned for the same event.

As discussed earlier independent resource policies have a different policy state. Therefore, if one policy allows a limited amount of tickets and another policy allows all access by setting `allowAll`, then access is still limited by the first policy.

Local tickets have to be revoked after all handlers for the specific event have been executed. The PhoneWrap event handler inserts a callback method into the HTML message queue by calling `setTimeout` with 0 seconds. Once all handlers for this event have been executed, JavaScript fetches the next element in the message queue and executes PhoneWrap's callback which sets the number of local tickets to 0.

The JavaScript function `dispatchEvent` can be used to generate an artificial event. A malicious app could use it to simulate user interaction and generate an arbitrary amount of tickets without real user interaction. To guarantee that tickets are only generated for real user interaction, PhoneWrap also wraps `dispatchEvent`. The wrapped version first disables ticket generation, then generates the artificial event and re-enables tickets after the event has been processed. The only other JavaScript API manipulating event handlers is `removeEventListener` which deletes all the event handlers for one DOM node. Even though the app could remove the PhoneWrap event handlers using this function, this would only generate fewer tickets, because button policies only generate tickets, and prevent the functionality of the app. The security of the policy enforcement is not threatened. In particular, the JavaScript code cannot extract the PhoneWrap event handler and call it without user interaction.

4.3.2.1 Special input elements

Motivated by real world apps, the button policies have been extended by two behaviour patterns which go beyond the formal specification of ITPs.

Some real world examples like "TrackMyVisit" make the user aware of the resource consumption by displaying a *confirmation dialog*. Pressing the "Emergency" button first brings up a confirmation dialog and the message is only sent if the user presses "Ok". Since confirmation dialogs are not part of the DOM tree, PhoneWrap does not receive events for the dialogs. To incorporate dialogs into policies, PhoneWrap

instruments the dialog APIs and wraps the callback functions of each dialog. If a button policy specifies a list of captions in the `confirm` parameter, PhoneWrap only *reserves* the specified number of tickets for such an event, rather than granting them. If in the subsequent dialog a button with one of the specified captions is pressed, the reserved tickets are granted. If the user presses a dialog button not specified in the `confirm` list, PhoneWrap deletes all reserved tickets. In this case, PhoneWrap can implement flow-sensitive policies.

Other apps consume resources dependent on a user selection, for example, send messages to each contact the user has marked in a list of checkboxes (see Section 4.5.3.8 for a real-world example). In this case, the consumption depends on the number of selections. In PhoneWrap's button policies, a UI element can be identified as a *checkbox* which instructs PhoneWrap to grant tickets when the box is checked and revoke the tickets when the check is removed.

4.3.3 Properties

To show that PhoneWrap enforces ITPs, the assumptions for the policy enforcement need to be verified. Due to the lack of a formal execution model of JavaScript apps, they are justified informally:

Claim 4.3.1. *The application code cannot change the state of the policy.*

JavaScript's scope protects the local variables of the policy function, including the policy state, against access from outside the policy function.

Claim 4.3.2. *All calls to the original function f are replaced by the wrapped version $wrap_{pol}(f)$.*

The reference f to the critical function is overwritten by the wrapped version in the JavaScript environment. Every time the application code calls f, the wrapped version is called instead. The PhoneWrap script is inserted at the top of the header of the main HTML file. Therefore, it is executed before any other script and overwrites all API references specified in the policy before the application code can store a local copy of the original reference. APIs made available after the PhoneWrap script has been executed are either discovered using the `deviceready` function or by the `MutationObserver`. In both cases PhoneWrap registers the listener before any other listener is registered. This ensures that PhoneWrap's listener is executed before any other handler and that the wrapping is executed before the application code can access the unwrapped API.

Claim 4.3.3. *The application code cannot access the original API directly.*

The original references are protected like the policy state. Due to JavaScript's dynamic scoping all calls to the original function are rerouted to the wrapped API, even for internal calls in JavaScript libraries. Note, however, that only the APIs specified in the policy are guarded. PhoneWrap must assume that all critical APIs are specified there.

Claim 4.3.4. *Tickets are only generated by the specified user interaction.*

The only PhoneWrap method (apart form potentially the deny behaviour) which increases the ticket count is the event handler. According to the HTML standard, the only way to execute an event handler without user interaction is the function `dispatchEvent` which is wrapped by PhoneWrap so it cannot generate tickets.

Furthermore, the PhoneWrap script is executed first and registers the first listener at the root node. Each event is first passed to the root node and the JavaScript standard specifies that event handlers are executed in the order of registration. Therefore, the tickets for an event are generated before the application code potentially uses the tickets for the functionality triggered by the event.

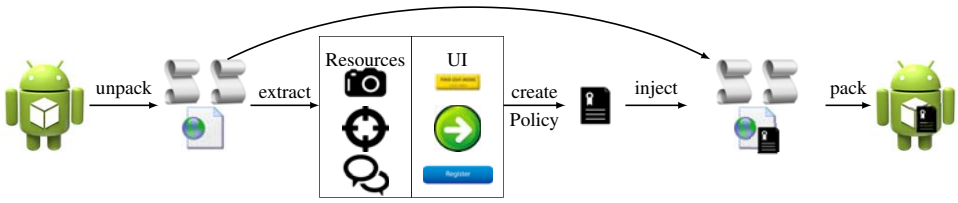
Corollary 4.3.5. *PhoneWrap implements an ITP and therefore enforces the bound specified in Section 4.2 on the resource usage of the wrapped app.*

PhoneWrap can inject multiple wrapper scripts into the same app if either the user wants to guard multiple different resources or multiple parties (developer, distributor, device admin, user, ...) have different policies for the same resource. In this situation, multiple wrapper scripts are inserted into the DOM tree, each with it's own policy definition and policy scope. In case a critical API with multiple separate wrappers is called, the original API is called only, if all policies allow access individually.

Since each wrapper is executed in its own function scope, the different wrappers cannot access each others' policy state. Due to this fact, the final decision whether to execute the original API is independent of the order in which the policies are injected into the app. In this way PhoneWrap, is modular. However, different injection orders might result in different deny behaviour. If an outer policy is violated the resource call is replaced by its deny behaviour. Since this might not call the wrapped function, deeper nested policies are not evaluated and their deny behaviour is not activated.

4.4 Policy injection

The injection process consists of the following steps:



The tool chain[2] of PhoneWrap supports each of the 5 steps:

1. The script `m10_unpack` unpacks an app package downloaded directly from the Google Play Store to access the JavaScript code.

2. The package analysis tool `m20_analyse_apk` extracts all necessary information for the policy creation from the configuration files. This includes the accessed resources (via the PhoneGap plugins and permissions) and the elements of the user interface which might perform the resource consuming functionality.

3. Users not familiar with JavaScript can write the policy using PhoneWrap's HTML

[2]Source code has been made available: `github.com/DFranzen/PhoneWrap`

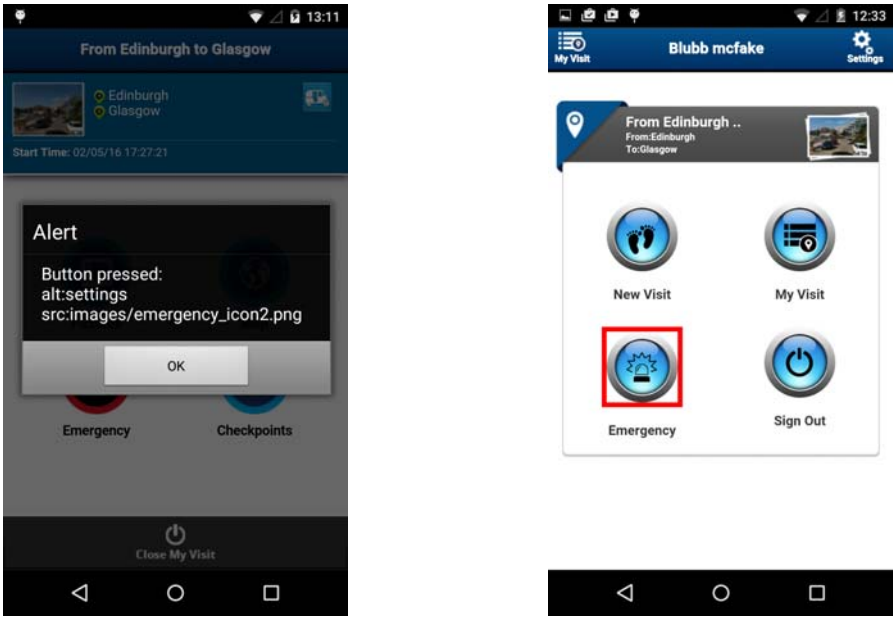
form `m25_createPolicy.html` shown in Figure 4.1. It provides the available choices for each parameter and embeds the policy into full English sentences.

4. The script `m30_policify` automatically inserts the wrapper script and a provided policy into the main HTML folder and links it into the main DOM tree of the app.

5. Finally, with `m90_pack_install` the app is packaged back into a standard Android app package. Since PhoneWrap altered the source code, this new package has to be re-signed after injection to be installed on an Android phone.

The only step that requires human action is the *policy creation*. The policy author has to specify the guarded APIs and specify the UI elements for the button policies. PhoneWrap assists with all tasks of this process. The analysis script `m20_analyse_apk` compiles a list of plugins and permissions included in the app. For the app "TrackMyVisit" from Section 3.1 this list contains the plugin `org.apache.cordova.plugin.SmsSendingPlugin`. PhoneWrap does not infer the critical APIs for each plugin automatically, but a table of critical APIs for the most common plugins is provided with the PhoneWrap source code. In the example, the identified plugin provides the JavaScript API `smsplugin.send` and the Java API `SmsPlugin.SEND_SMS`.

Figure 4.4 TMV - Policy Creation



(a) Instrumentation **(b)** Policy Verification

To choose the correct properties for the button policies, PhoneWrap instruments

the original app to display the properties of each button during normal interaction (see Figure 4.4a). To check the correctness of the button policies, the same instrumentation can display each affected button in a red frame (see Figure 4.4b). Thanks to these features, a PhoneWrap policy can be created for an app without knowledge of the app's source code or implementation details of the enforcement method.

Android app packages are signed by the developer of the app and this signature is validated by Android during installation to preclude modification of the app by third parties. Since PhoneWrap modifies the code the signature by the original developer is invalidated. While packaging the injected app into an Android package, PhoneWrap re-signs the app with the key of the policy injector. Assuming the original signature is verified, the new signature vouches for the fact that the original signature was not violated before injection and that the policy was inserted. Instead of trusting the developer of the app, now the trust lies with the user injecting the PhoneWrap policy as well as their trust in the original developer. Sometimes, functionalities for interaction between different apps depend on the signatures of the two apps. Therefore, it is advised to re-sign apps that were signed with the same original key with the same policy key.

An automated script `mall` executes all scripts mentioned above in the correct order to directly insert a given policy into the package downloaded from the app store.

4.5 Evaluation

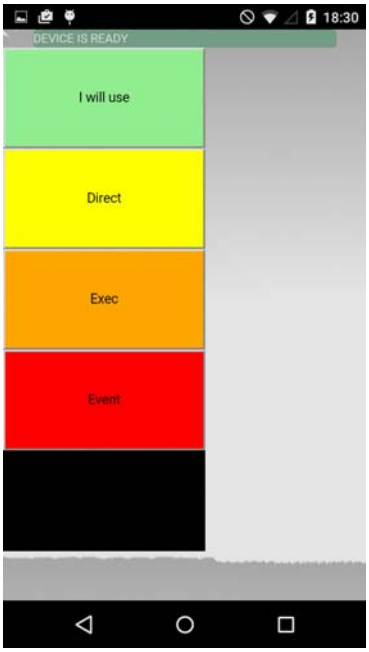
The evaluation of PhoneWrap on real apps aims to answer the following questions:

1. Does PhoneWrap restrict the wrapped app's resource consumption to the specified bounds?
2. Is the runtime overhead of the enforcement acceptable?
3. Is the policy specification fine-grained enough to describe the benign resource behaviour of real-world apps?
4. Can the appropriate policy for an app be identified with the provided tools?

4.5.1 Enforcement tests

To evaluate question 1, the app in Figure 4.5 is crafted to circumvent PhoneWrap's enforcement. The critical resource in this experiment is the vibration feature, since the

Figure 4.5 Experimental App




```
policy = {
    mperms:0,
    buttons:[
      {
        cond: {
          id: "USE"
        },
        mperms: 1,
        match: "exact",
      }
    ],
    guard: ["navigator.vibrate"],
    guard_exec: ["Vibration.vibrate"],
    deny: function(){alert("Policy:Denied")}
}
```

effect of resource access is noticeable immediately and the corresponding plugin offers a JavaScript API as well as a Java API.

The UI of this app consists of 4 buttons each trying to access the resource in a different way. The first button (green) calls the JavaScript API and serves as the control test to ensure legitimate resource access is not prohibited by the injected policy. The second button (yellow) uses the same method and represents standard non legitimate access. The third button (orange) calls the Java API via the `exec` function and the bottom button (red) sends an artificial click event to the green button. Each button also changes the colour of the panel at the bottom (black in Figure 4.5) to simulate the resource independent functionality. Furthermore, to represent resource access without user-interaction, the app calls the vibration API scheduled every 30s.

The policy in Figure 4.5 guards the vibration APIs and grants 1 ticket for each click on the green button (identified by its id “USE”). After policy injection, the vibration of the yellow, orange and red buttons and the scheduled vibration are successfully prohibited. The behaviour of the colour of the bottom panel is not affected by the policy, showing that resource independent functionality is preserved. The behaviour of the green button is unchanged which shows that the bounds are enforced as specified.

For a second experiment with the same app, the green button was greyed out (using CSS classes) for the first 30s after launch. Messaging apps, for example, often grey out the “Send” button before the user has entered the message text or recipient. The

policy was amended to only generate tickets for green buttons. As expected, vibration was denied during the initial greyed-out period but allowed after the button colour is changed to green. This shows that PhoneWrap reacts correctly to changes in the UI.

To evaluate question 2, the runtime overhead of the enforcement, the APIs to start and pause audio playback were chosen as the resource. They can be called in quick succession and do not require user interaction. Therefore, they are suitable for automated tests. The app for this evaluation contains of only one button which starts and pauses the playback of an audio file 1000 times as quickly as possible. As a result the app returns the measured times for these 2000 API calls. This app was then injected with a policy guarding the JavaScript APIs of the audio plugin and allows 2000 tickets per button click.

Figure 4.6 Running times, 2000 API calls each

(a) JavaScript APIs

Test	Original app	Injected app
1	4598 ms	4823 ms
2	4500 ms	4540 ms
3	4527 ms	4719 ms
4	4599 ms	4558 ms
5	4616 ms	4883 ms
ø	4568 ms	4704 ms

(b) Java APIs via `exec`

Test	Original app	Injected app
1	5072 ms	5131 ms
2	5151 ms	5227 ms
3	5057 ms	5117 ms
4	5035 ms	5411 ms
5	5130 ms	5182 ms
ø	5089 ms	5213 ms

To eliminate noise, the app was executed 5 times with the resulting times in Figure 4.6a. The overall overhead amounts to 136ms on average. This corresponds to a 3% runtime overhead of API calls or less than 1ms per API call.

The results of the same test with direct calls to the Java APIs via the `exec` bridge are shown in Figure 4.6b. The overall overhead of 124ms is similar to the test with JavaScript API calls. The longer runtimes in this test probably originate from the fact that this test had to manage the parameters for the API calls manually and did so less efficiently than the plugin code. Due to the longer overall time, the overhead reduced to 2.4%. This also shows that other implementation details have a higher impact on the overall runtime than PhoneWrap's enforcement.

Apart from the instrumentation enforcing the policy, PhoneWrap also adds time to the launch of the app to initialise the enforcement. The initialisation for the audio API in JavaScript and Java took between 8 and 20ms. The interaction policies added about 3ms to the handling of each button event. In summary, the overhead of the enforcement

method during these test does not significantly influence the performance of the app.

4.5.2 Real-world apps

To answer questions 3 and 4, PhoneWrap was tested on a set of real-world apps downloaded directly from the Google Play Store. In total, 8757 apps used for the evaluation in [67] were available for download during November and December 2014.

Figure 4.7 Categories of SMS apps

	# apps	PhoneGap version	
a	5400	$\geq$2.0	PhoneWrap applicable
b	1443	$<$2.0	
c	1914	N/A	Configuration file not as expected
total	8757		

The analysis script could identify (see Figure 4.7) and parse the PhoneGap configuration file in 6843 (a + b, 78%) of these apps. The necessary information about the used resources and the main source files could be extracted making policy creation and injection possible. The 1443 apps in category b are written with PhoneGap prior to version 2.0. Since the structure of these configuration files differs significantly from the current version, PhoneWrap had to be slightly adapted for these apps. It now recognises these versions automatically. The inspection of a small sample of the remaining 1914 apps (c) showed that they either use very early versions of the PhoneGap library, where the plugin structure was not fixed yet, or include the PhoneGap library without using the PhoneGap framework. In the first case the PhoneWrap wrapping script can probably be used to guard the critical resource in a majority of the apps, but the resource accessing APIs have to be identified in the code manually.

4.5.2.1 Real world evaluation setup

In this evaluation, the messaging service was chosen as resource, since it is used in real-world apps with immediate and quantifiable effects on the privacy and assets (allowance, phone bill) of the user. Apps accessing the messaging service can easily be identified by filtering for the permission `SEND_SMS`. Concerning the implementation, PhoneGap does not provide an official plugin to access the messaging service. Instead, multiple different third party plugins are available. This way, PhoneWrap is exposed

to different plugin implementation techniques during this experiment. The candidate apps chosen from the set of 6843 PhoneGap apps were evaluated on the test device (Motorola Moto G, Android 5.0.2) by applying the following steps:

1. Manually explore the UI of the app, monitor sent messages in the log and note the triggering user interaction sequences.

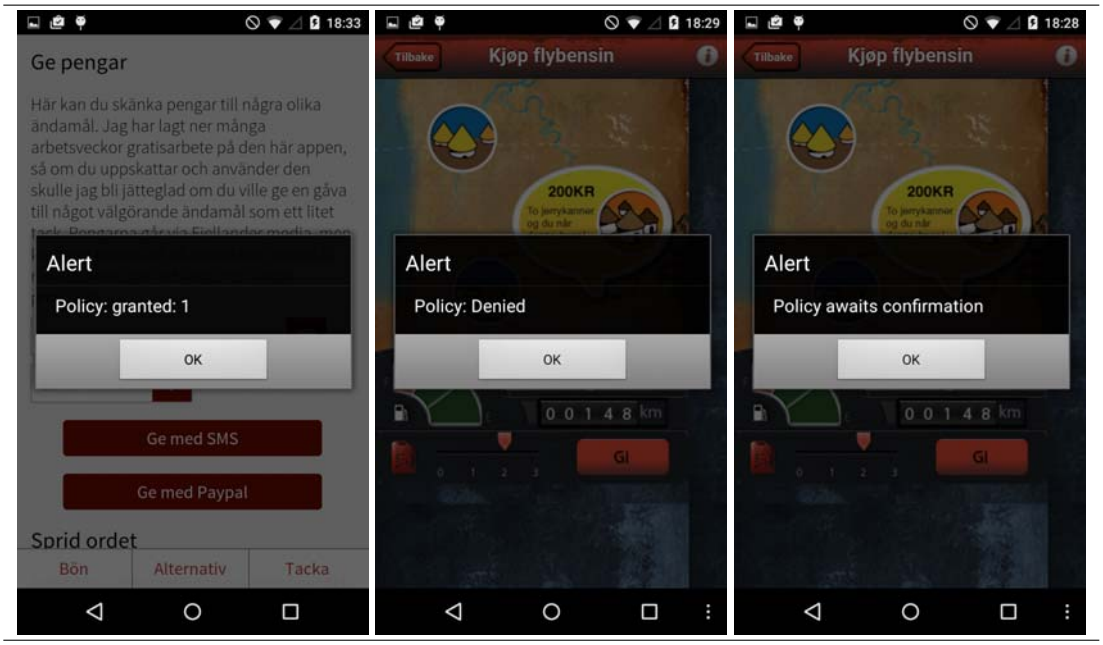
2. Instrument the app to identify the properties describing these interaction sequences.

3. Create a policy generating the exact amount of tickets needed to perform the expected behaviour for each legitimate interaction.

4. Inject this policy into the original package and manually evaluate the behaviour of the modified app.

The wrapper script used in this evaluation is amended to provide feedback about the correctness of the policy enforcement: pop-up messages report every change of the policy state as seen in Figure 4.8 and the deny behaviour replaces the resource request by the pop-up message “Denied”. This way, each state change of the policy can be verified against the model of ticket-based policies.

Figure 4.8 Policy status output



4.5.2.2 Messaging in PhoneGap

Out of the 6843 eligible apps only 59 apps request the permission to send SMS. The Android permission model prohibits all remaining apps to send SMS. The 59 apps split into categories as shown in Figure 4.9. The apps in category a could not be examined,

Figure 4.9 Categories of apps with access to messaging service

Category	#Apps	Description	Policy
a	16	not testable (non-Latin alphabet or account required)	
b	16	no message plugin recognised	retract permission
c	13	no messages sent during UI experiments	deny-all policy
d	4	send messages via intent	deny-all policy
e	10	candidates	individual policy
total	59		

because their user interface is in non-Latin letters or they require an account which could not be easily created. The behaviour of apps in this category is not known and, hence, no policy could be created. A normal user of these apps could, however, describe the behaviour and create policies accordingly. Apps in category b do not include a recognisable SMS plugin. They are either over-privileged or use a nontransparent third party SMS plugin. The PhoneWrap tools can be used to unpack the app, remove the SMS permission from the Android Manifest and repack the app. Since PhoneWrap cannot guard the resource accesses individually, quantitative or interaction-dependent policies cannot be fitted. Category c contains apps which did not send any messages during the manual examination. Assuming the UI was examined sufficiently, the apps should be fitted with a deny-all policy. In contrast to category b, the resource consuming APIs are known and the PhoneWrap deny-all policy described on page 52 can be inserted. This way, PhoneWrap can define a more flexible counter-action should the guarded API be accessed. The redaction of the permission used for category b always terminates the app with a security violation. Category d contains 4 apps which contain a messaging plugin, but all discovered messages were sent through intents to the Android messaging app instead. This method requires the user to explicitly press the “Send” button in the familiar messaging UI and the app itself does not require the SMS sending permission. Thus, the apps in this category are certainly over-privileged and PhoneWrap can deny all access as in category c to prevent additional resource consumption in the background. This leaves 10 apps which actually use PhoneGap to

send SMS and can be fitted with a meaningful quantitative policy.

In the categories a, c, d and e, 19 different message sending plugins were identified (see Figure 2.11 for details). For 12 of them the full source code is available from which the resource consuming JavaScript and Java APIs were identified. For the remaining 7 the JavaScript source code can be extracted from the app package and the used part of the Java API can be reverse engineered. The Java part might contain further API functions only accessible via `exec`, but such hidden APIs were not found in any of the plugins with available source code. This increases the confidence that the APIs for unknown plugins can be extracted from the app itself. Nevertheless, to defend against hidden Java APIs, PhoneWrap could instead employ a white-list and deny all unknown Java APIs. The 5 different SMS plugins included in the set of candidates (category e) are all available in full source code.

4.5.2.3 Evaluation results

PhoneWrap was successfully applied to all 10 candidates. Figures 4.10 and 4.11 show the functionalities and injected policies for each app. The expected behaviour was easily identified by using the app for a few minutes. After policy injection, the allowed behaviour was performed identically to the original app. The displayed state changes show that the tickets are managed as specified in the policy.

Within the 10 apps, PhoneWrap was able to demonstrate most of its features: the app Servicehours (10) demonstrates the need for the `checkbox` policy and several apps (6-9) justify the `confirm` policies. In TrackMyVisit (9) the `local` tickets are necessary to tighten the resource consumption bound. On the other hand, the policy for Servicehours (10) needs non-local tickets, since the ticket generation and ticket consumption are triggered by different events. The other apps (1-8) use all granted tickets immediately and, therefore, local and global tickets are equivalent. The policies below grant local tickets as the fail-safe method.

The apps Tidegarden (5) and MAF (7) send charged premium messages. Before such a message is sent, Android warns the user in a confirmation dialog (see Figure 4.13 right). This dialog is outside the scope of PhoneWrap, since PhoneWrap enforces its policy on the application level. As a consequence, the system dialog is displayed when the PhoneWrap policy grants the message and a ticket is used even if the user cancels the Android dialog. If the PhoneWrap policy denies the message, the dialog is suppressed as part of the API call.

In the app Glassmester (8), PhoneWrap was able to prohibit unwanted resource

consumption. This app displays a confirmation dialog before the message is sent. However, the message is sent even if the user presses the "Cancel" button! The injected PhoneWrap policy grants the message only if the "OK" button is pressed. Thus, PhoneWrap isolates a bug in a real world app.

Figure 4.10 App functionality

App no	App functionality	Resource consuming interaction
1	Manage the Insurance (Read policy, Get a quote, Report Accident)	Contacts → Text → SEND TEXT
2	Remote control appliances on a boat via SMS	<Any device button> → <Any command button> → Send
3	Remote control cameras via SMS	<Any device button> → <Any command button> → Send
4	Remote control devices via SMS	<Any device button> → <Any command button> → Send
5	Display prayer texts for each day	Tacka (cover) → Ge med SMS (Donate via SMS)
6	Remote control devices via SMS	<Any envelope button> → Send
7	Inform about and support MAF	Kjop flybensin (buy jetfuel) → <Any stage> → Gi (Donate) → Ok
8	Order and manage glass reparations	Befaring (Inspection) → SMS → Ok
9	Create journey logs	Emergency OR New Visit → Emergency
10	Business Search and Information Access	Gear → INVITE FRIENDS → Envelope

Figure 4.11 Apps and their Policies

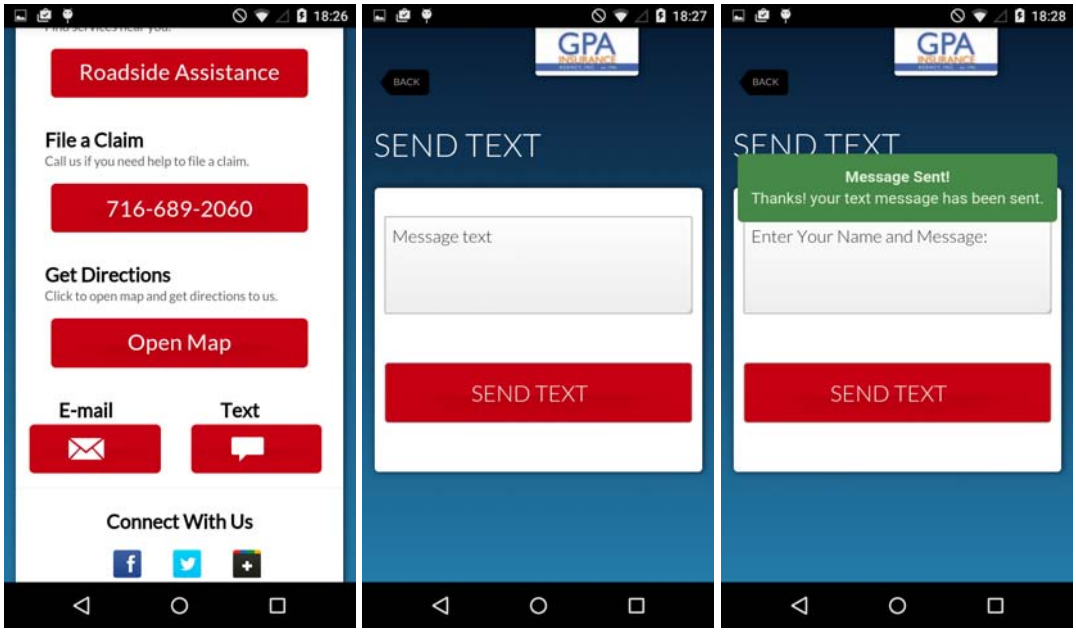
	App (version, versionCode)	Button Condition	Tickets	Policy features
1	com.GPAInsurance.myinsurance.apk(1.0, 2)	Caption: "Send Text"	1	
2	nu.fdp.Boatsteward.apk(1.5, 6)	id: "btnSendTheMessage"	1	
3	nu.fdp.optimaxx_gsm.apk(2.6, 26)	id: "btnSendTheMessage"	1	
4	nu.fdp.Sms_RC.apk(4.1, 19)	Caption: "Send"	1	
5	se.fjellandermedia.tidegarden.apk(3.1, 310)	Caption: "Ge med SMS" id: "donateSMS"	1	
6	nu.fdp.Sms_RC_Mini.apk(1.7, 8)	src: (ends with) "/pict/send.png"	1	confirm: "Send"
7	no.idium.apps.maf.apk(1.0.1, 68)	id: "stage_Gi"	1	confirm: "Ok"
8	no.idium.apps.apk(1.0, 52)	class: "sms_small"	1	confirm: "OK"
9	myzealit.TMV.apk(2.2, 22)	src: (ends with) "emergency_icon.png"	3 (local)	confirm: "Yes"
10	com.ServiceHours.ServiceHours.apk(1.3.3, 8)	name: "contactnumber"	1 (global)	checkbox

4.5.3 Policies added to the real world apps

4.5.3.1 GPA Insurance

The app GPA Insurance (1) shows the easiest resource behaviour: the user can contact the issuing company via SMS triggered by a button. After clicking on the speech-bubble button in the "Contacts" menu, the app provides a "SEND TEXT" screen. The user can enter the content of a message and send it via the red "SEND TEXT" button. The PhoneWrap instrumentation reveals that the caption "Send Text" can be used to

Figure 4.12 GPA Insurance



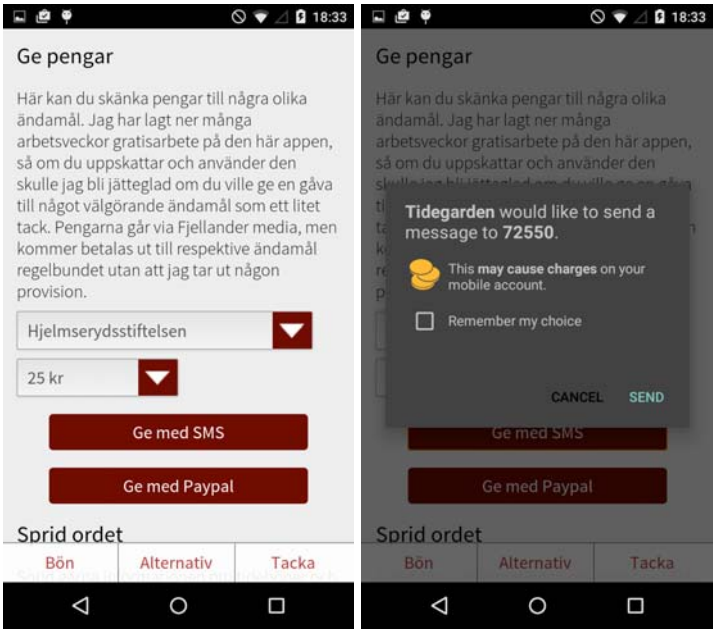

```
policy = {
        mperms : 0,
        buttons : [
            {
                cond: { value:"Send Text" },
                mperms:1,
                match: "exact",
            }
        ],
        guard: [
            "window.plugins.sms.send",
            "SmsPlugin.prototype.send"
        ],
        guard_exec: [ "SmsPlugin.SendSMS" ],
        deny: function(){alert("Policy: Denied")}
    }
```

specify the activating button. The policy generates one local ticket for each click on this button.

4.5.3.2 Tidegarden

The Swedish app “Tidegarden” (5) allows the user to send a message donation via the button “Tacka” (“cover”). The user can select the organisation they want to donate to and by clicking on “Ge med SMS” (Donate via SMS) a premium SMS for the specified amount is sent. The policy generates 1 ticket for the button with the id “donateSMS” and the caption “Ge med SMS”.

Figure 4.13 Tidegarden



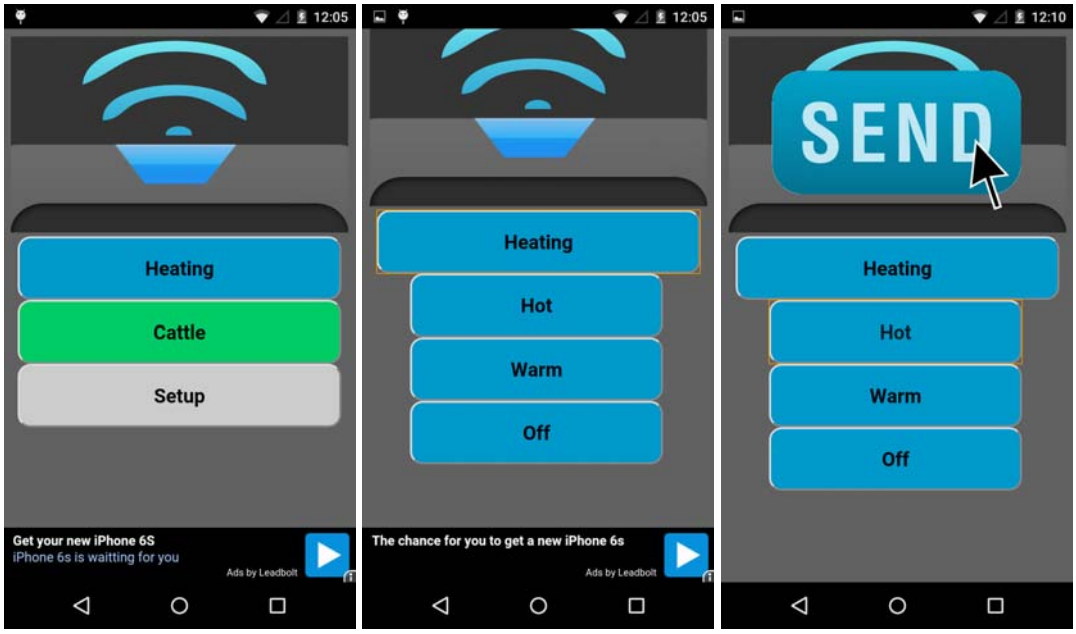

```
policy = {
        mperms : 0,
        buttons : [
            {
                cond: {
                    value: "Ge med SMS",
                    id: "donateSMS"
                },
                mperms:1,
                match: "exact",
            }
        ],
        guard: [ "smsExport.sendMessage" ],
        guard_exec: [ "Sms.sendMessage" ],
        deny: function(){alert("Policy: Denied")}
    }
```

4.5.3.3 Boatsteward, Optimaxx, Sms RC

The app Boatsteward (2) allows the user to specify a remote for boat appliances with message receiving capabilities. Via the setup menu the user can specify the phone numbers of up to 10 devices and 6 command texts per device. When the user selects a device and a command the sending button appears which sends the specified command to the device's phone number. The send button does not have a caption but can be identified by its id "btnSendTheMessage":

The inspection of the HTML document of this app reveals that it uses JavaScript to insert the PhoneGap framework script dynamically to make the code cross-platform compatible. Since PhoneWrap listens for dynamically injected scripts via the MutationObserver standard, it can wrap the APIs as soon as the library is loaded.

Figure 4.14 SMS Remote apps




```
policy = {
        mperms : 0,
        buttons : [
            {
                cond: { id:"butSendTheMessage" },
                mperms:1,
                match: "exact",
                local: true
            }
        ],
        guard: [ "smsplugin.send" ],
        guard_exec: [ "SmsPlugin.SEND_SMS" ],
        deny: function(){alert("Policy: Denied")}
    }
```

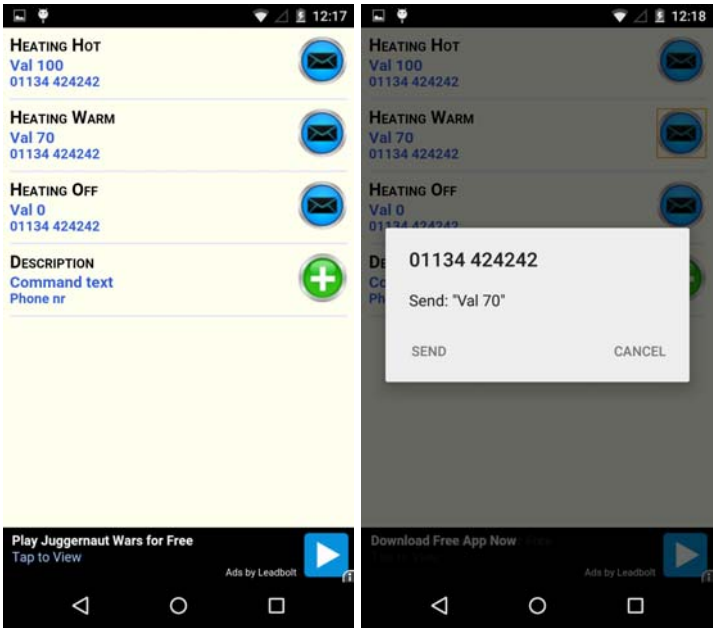
The app Optimaxx GSM (3) uses the same code with a slightly redesigned user interface. The policy for Boatsteward can be reused directly for Optimaxx. For the app Sms RC (4) the policy has to be slightly modified to describe the button by its caption “Send” instead.

4.5.3.4 Sms RC Mini

App 6 is a completely revised version of the apps 2-4. It uses the same SMS plugin but displays a separate envelope button to send the message for each task. The buttons can be uniformly described by their icon path ending in “/pict/send.png”.

This app inserts the whole DOM tree dynamically except for the bare skeleton. Since PhoneWrap’s listeners are registered at the DOM root, it can be attached to the initial DOM tree, therefore detecting all added notes automatically. Additionally, the

Figure 4.15 Sms RC Mini



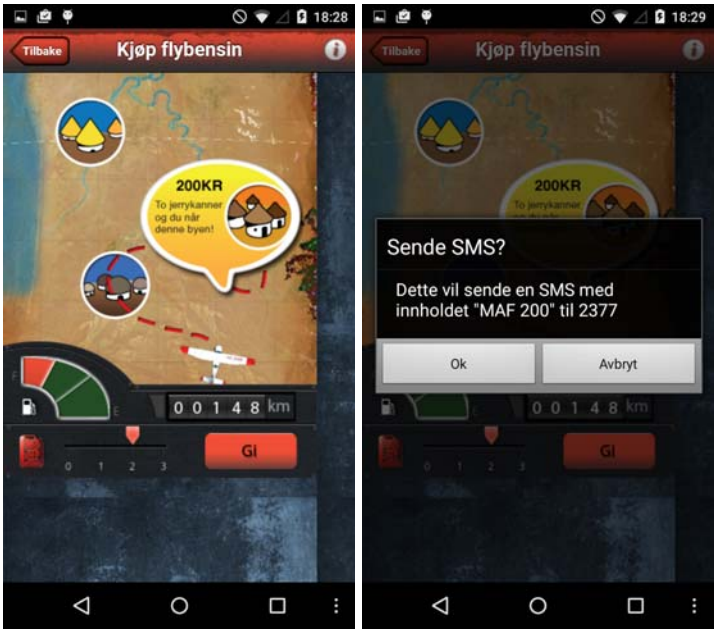

```
policy = {
        mperms : 0,
        buttons : [
            {
                cond: { src:"/pict/send.png" },
                mperms:1,
                match: "ends",
                confirm: ["Send"],
                local: true
            }
        ],
        guard: [ "smsplugin.send" ],
        guard_exec: [ "SmsPlugin.SEND_SMS" ],
        deny: function(){alert("Policy: Denied")}
    }
```

app also uses a confirmation dialog before the message is sent. Here, the `confirm` feature of PhoneWrap is used to only allow tickets when the user clicks the confirming "Send" button

4.5.3.5 MAF

The app MAF Norge (7) is the Norwegian app of the charity Mission Aviation Fellowship which provide air support for remote areas in need all over the world. The app sends premium SMS in the Menu "Kjop Flybensin" (Buy jet fuel) for each click on "Gi"(give) to donate for their missions. The resource consuming button can be identified by its id "stage_Gi". Again, the user needs to confirm the message by clicking the "Ok" button in the subsequent dialog.

Figure 4.16 Missions Aviation Fellowship




```
1      policy = {
2          mperms : 0,
3          buttons : [
4              {
5                  cond: { id:"stage_Gi" },
6                  mperms:1,
7                  match:"exact",
8                  confirm: ["Ok"],
9                  local: true
10             }
11         ],
12         guard_require: [ "cordova/plugin/sms.send" ],
13         guard_exec: [ "SmsPlugin.SendSMS" ],
14         deny: function(){alert("Policy: Denied")}
15     }
```

4.5.3.6 TrackMyVisit

The functionality of the app TrackMyVisit (9) has already been discussed in Section 3.1. After the user clicks the “Emergency” button on the main screen and confirms with “Yes” the app sends a message to up to 3 specified contacts. The emergency button can easily be identified by its icon ending in “images/emergency_icon.png”. A second emergency button exists in the trip specific menu with the icon “images/emergency_icon2.png”.

This app is an example where multiple tickets are generated per button click, in this case, 3. It also illustrates the need for the `local` parameter. The app is granted 3 tickets, but depending on the setup, might not send all 3 messages. The remaining tickets could then be used for hidden or unwanted behaviour in the background at a later stage. To prevent that, PhoneWrap revokes the remaining local tickets after all

Figure 4.17 TrackMyVisit

```
1      policy = {
2         mperms : 0,
3         buttons : [
4             {
5                 cond: { src:"images/emergency_icon.png" },
6                 mperms:3,
7                 match:"ends",
8                 confirm: ["Yes"],
9                 local: true
10            },
11            {
12                cond: { src:"images/emergency_icon2.png" },
13                mperms:3,
14                match:"ends",
15                confirm: ["Yes"],
16                local: true
17            }
18        ],
19        guard: [ "smsplugin.send" ],
20        guard_exec: [ "SmsPlugin.SEND_SMS" ],
21        guard_require: [ "cordova/plugin/smssendingplugin.send"
              ],
22        deny: function(){alert("Policy: Denied")}
23    }
```

handlers for this event have been executed.

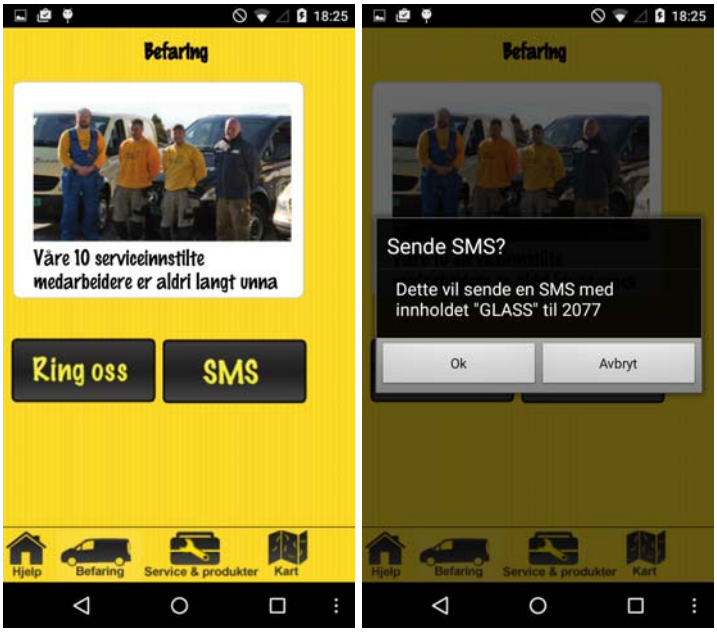
4.5.3.7 Glassmester

The Norwegian app Glassmester (8) allows a customer to send an SMS to the customer service crew via the button sequence “Befaring” and “SMS”. The app shows a confirmation dialog “Sende SMS?” (Send SMS?), where the user can select “Ok” or “Avbryt” (Cancel). However, if the user selects cancel, the message is sent anyway. This contradicts the expected behaviour. PhoneWrap restricts the app by specifying the button with the class “sms_small” to generate tickets only when confirmed with “Ok” in the following dialog. Through the injection of the policy the faulty behaviour could be corrected.

4.5.3.8 Servicehours

The app Servicehours (10) can be recommended to friends via SMS by clicking on the gear button in the top left corner and then selecting “Invite Friends”. The app presents the user with a list of all contacts stored on the phone which the user can select individually using provided checkboxes. With the checkbox policy, PhoneWrap only allows the app to send one messages per selected friend. The policy identifies the checkboxes by their property `name` which is set to `contactnumber`.

Figure 4.18 Glassmester



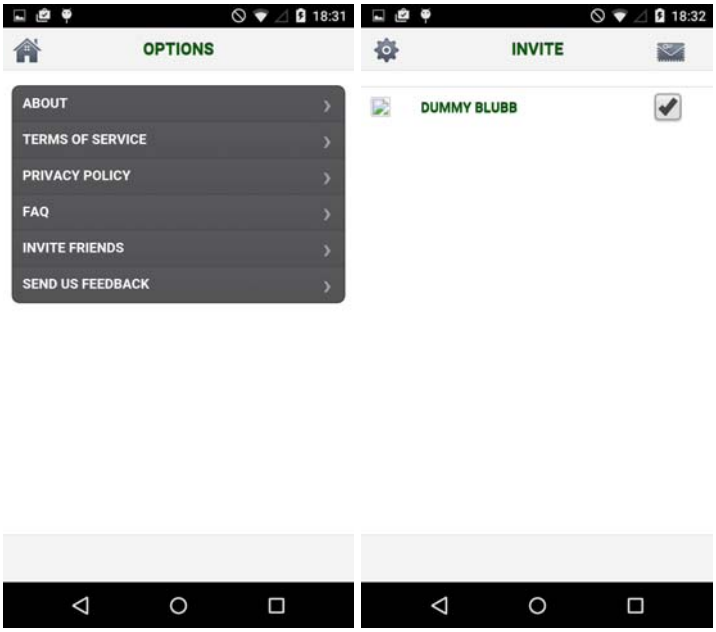

```
policy = {
        mperms : 0,
        buttons : [
            {
                cond: { class:"sms_small" },
                mperms:1,
                match:"exact",
                confirm: ["Ok"],
                local: true
            }
        ],
        guard_require: [ "cordova/plugin/sms.send" ],
        guard_exec: [ "SmsPlugin.SendSMS" ],
        deny: function(){alert("Policy: Denied")}
    }
```

This app demonstrates the necessity for the checkbox feature of PhoneWrap. It also illustrates the need for global tickets: the click on the checkbox generates the tickets, but the event handler of the envelope button uses them. The tickets have to stay valid until the second button is pressed, which is after the event handler for the checkbox has already finished.

4.6 Discussion

The system PhoneWrap injects quantitative policies into real world PhoneGap apps to limit their resource consumption. To enforce those policies during runtime, PhoneWrap wraps the resource consuming APIs and monitors how often they are called. If the app exceeds the stated bound, a deny behaviour is executed instead of the resource access.

Figure 4.19 Servicehours




```
1     policy = {
2         mperms : 0,
3         buttons : [
4             {
5                 cond: { name:"contactnumber" },
6                 mperms:1,
7                 match:"exact",
8                 checkbox: true
9             }
10        ],
11        guard: [ "smsplugin.send" ],
12        guard_exec: [ "SmsPlugin.SEND_SMS" ],
13        guard_require: [ "cordova/plugin/smssendingplugin.send"
              ],
14        deny: function(){alert("Policy: Denied")}
15    }
```

Additionally, PhoneWrap listens for events on specific UI elements and allows the resource access required by the triggered functionality. The system extracts all information needed to write the fine-grained policies and can insert the policy into standard app packages automatically. In tests with real apps PhoneWrap successfully injected appropriate policies into 10 selected apps restricting the messaging behaviour. In one case PhoneWrap was able to prevent the app from sending unexpected messages. From the whole example set of ~8000 apps, PhoneWrap can extract all needed information to create and inject a policy into 78% of the examined apps.

Some additional checks need to be implemented to make the wrapping impenetrable: PhoneWrap, as presented here, trusts the integrity of the PhoneGap library. An altered PhoneGap library could provide additional APIs to access native resources which are not guarded by PhoneWrap. By checking the library included in the app

against all known versions, for example by comparing their hashes, PhoneWrap can make sure all provided APIs are guarded. The same is true for the PhoneGap plugins. PhoneWrap needs to make sure that the Java classes packaged in the PhoneGap app are genuinely implementing the expected API and not more. With a whitelist of known plugin versions PhoneWrap can, for example, warn during policy injection if unknown plugins are included. The required data is easily obtained from the plugins' GitHub repositories.

In PhoneWrap's current version it is also not difficult for an app to discover that it has been injected with a policy: the `toString` function and others return different values for the original method and the wrapped method as shown in Figure 4.20. Since

Figure 4.20 Before and After Policy Injection

(a) Without Policy

```
  > navigator.notification.alert.toString():
1 function (message,completeCallback, title, buttonLabel)
2 {
3     var_title = (title || "Alert");
4     var_buttonLabel = (buttonLabel || "OK");
5     exec(completeCallback,null, "Notification", "alert", [message,_title,
          _buttonLabel]);
6 }
```

(b) With Policy

```
  > navigator.notification.alert.toString():
1 function() {
2     return exec_guarded(orig,this,arguments);
3 }
```

the original function cannot be extracted from this output, this fact does not compromise the security of the system, although it might give an attacker the opportunity to replace the resource by an equivalent one (e.g. by using a network service instead of SMS to send out private data). It is possible to wrap the `toString` functions and other methods to behave equivalently on original APIs and wrapped APIs. However, finding and overwriting all such functions is tedious work and does not obviously add to the security of the system.

A final modification to make PhoneWrap more usable is to integrate the information about the APIs for each resource shown in Section 2.2.2 into the analysis script. This way, PhoneWrap could recommend which resources and APIs should be guarded based on the included plugins and permissions.

The injection of PhoneWrap is specific to Android. However, the wrapping script

is not. Created policies will be enforced for all PhoneGap apps independent of the operating system. With policies adapted to browser specific resource and injected by existing browser plugins like UserScripts for Opera PhoneWrap could enforce ITPs also for web applications on desktops.

Taking inspiration from the frameworks like PhoneGap, the Electron framework by GitHub [44] provides a similar paradigm for Desktop computers. It packages HTML and JavaScript code into native applications which execute the JavaScript code in a platform built-in browser view. The plugins for this framework are provided similarly to PhoneGap. With a few adjustments the PhoneWrap enforcement script could be applied to such applications to guard the desktop specific resources of programs written with the Electron framework.

Chapter 5

Type systems for JavaScript

JavaScript presents some unique challenges for static analysis as discussed in Chapter 2.1. Nevertheless, there have been approaches to capture its behaviour with type systems. This chapter presents an overview of these systems and discusses their methods, their properties and individual advantages.

The aim of the discussion here is to survey common methods used in typing JavaScript and to identify a suitable system to be augmented with amortised annotations by the system AmorJiSe in Chapter 6.

5.1 Taxonomy of type systems

The results of a type system are expressed by the *type judgement*. For languages with strong updates, like JavaScript, it can be written in the following form:

$$\Gamma \vdash e : t | \Gamma'.$$

Here, e is the analysed expression; Γ is the type context which assigns types to the free variables occurring in e; t is the type assigned to e and Γ' is the new context reflecting all changes to the types of the variables made during the evaluation of e.

Depending on the type system, the type judgement can have different properties. The following sections illustrates some of these properties which are important for resource analysis. The code in Figure 5.1 is used as an example throughout this section.

Figure 5.1 Example program

```
1 function f(x) {
2   var o = {};
3   o.a = "hello";
4   if (x > 3) {
5     o.b = x;
6   }
7   o.a = o.a + " world";
8   return o;
9 }
```

5.1.1 Type expressivity

Types can be used to validate different aspects of JavaScript programs including absence of runtime errors [9], absence of restricted dataflow [49] or inclusion in a subset of the JavaScript language [28]. The shape of the resulting types depends on the aspect a system was designed to analyse. For example, λ_{JS} [49] proposes a single type `JS` expressing that the given expression is conform with a statically safe JavaScript subset. The types in JS_0^T [9] are more expressive and describe the structure of values. The latter is referred to as *data types* in the following. In the setting of session types [110] a type even describes the interaction between multiple systems.

Data types for JavaScript are constructed from simple types (`Bool`, `Int`, `String`...) and types for objects and functions. JavaScript *objects* map field names to values and object types mirror this by mapping all known field names to their field's type. Objects are often notated as *row types* $[field_1 : t_1, ..., field_m : t_m]$ indicating that an object contains the fields $field_1, ..., field_n$ and the value of the field $field_i$ has the type t_i. In practice, an object type only mentions a subset of the actually available fields of an object, since a type has to over-approximate all possible traces through a program. *Function types* $O \times t_x \to t_{ret}$ describe the structure of the function parameters t_x and the return value t_{ret}. The type O describes the structure of the implicit parameter `this` used in JavaScript to access to object `o` in an object method `o.m()`.

Example data types for the code in Figure 5.1 can be provided as

`f` : $[] \times$ `Int` $\rightarrow$ [`a` : `String`]	
`x` : `Int`	lines 2-8
`o` : []	line 2
[`a` : `String`]	lines 3,7-8
[`a` : `String`, `b` : `Int`]	line 5

The function type for `f` expects an empty object for the `this` parameter and an integer for the parameter `x`. The function returns an object with at least the `String` valued field `a`. The variable `x` has type `Int` throughout the whole program and `o` is an object with the string field `a` set in line 3 and additionally the integer field `b` in line 5.

5.1.2 Soundness

A type system is considered *sound* if the property obtained as result of the analysis is guaranteed for all possible executions of the analysed code. *Structural soundness* for a data type system holds, if for all results $\Gamma \vdash e : t | \Gamma'$ the type t describes the structure of all possible values of e. The type [`a` : `String`, `b` : `Int`] for `o` in line 5 of Figure 5.1 is sound, because for any trace reaching this line, the object `o` contains at least the `String` valued field `a` and the `Int` valued field `b`.

Even though some of the type systems discussed below do not have structural soundness as their primary goal, they all provide this property.

Soundness of a system is usually demonstrated either by a *formal* proof or by the evaluation of *benchmarks*. Formally proven sound systems provide some form of *formal semantics* modelling the evaluation of the considered JavaScript subset. They show that all possible values of any expression according to this model are correctly described by the resulting type. To expand the guarantees made by the type system onto real-world code, only the correspondence between the formal semantics and the real-world language needs to be validated. Instead, other systems demonstrate soundness by applying the system to a set of examples. The system is claimed sound if the result of the analysis holds true for all examples. While testing-based evaluation is able to validate systems which rely on relations between values too complicated to handle with a formal proof, it only provides a guarantee of soundness for the tested cases and the tested implementation.

In contrast to sound type systems, some type systems are designed to *estimate* the type unsoundly under certain conditions. In Figure 5.1, if the condition `(x>3)` is true in most cases, the analysis can estimate it to be true for all practical cases and, as a result, assume that the field `b` always exists. Such a type system is not sound as there exist traces through the function body returning values for which the structure demonstrated in the type does not apply. If sound analysis cannot get satisfying results, unsound methods are a way to provide a result which is true for most practical cases.

5.1.3 Coverage

The full JavaScript language has many constructs and features, including built-in libraries like `Math`, `Set` or `Regex` and strongly connected libraries the DOM API which provides the interaction between HTML and JavaScript. Most formal systems only consider a core of the full language in a similar manner as originally proposed for Algol by Reynolds [98]. Features not included in this core language are either *translated*, handled by a practical implementation *without formal proof* or left for *future work*. Translation is achieved by a *desugar* function, which maps non-covered features to semantically equivalent expressions in the covered language. However, the replaced expression is not guaranteed to be resource equivalent. In the example in Figure 5.1, the nested field lookup line 7 can be desugared into the two lines

```
7a   var tmp = o.a;
7b   o.a = tmp + "world";
```

This translation is semantically equivalent, however, requires the temporary variable `tmp` using additional memory space.

Since the desugaring method replaces an expression e by a semantically equivalent expression $desugar(e)$ the type judgement $\Gamma \vdash desugar(e) : t|\Gamma'$ implies $\Gamma \vdash e : t|\Gamma''$ with a Γ'' corresponding to Γ' in some way for most type systems. Therefore, data types obtained from a desugaring type system can usually be used equivalently to the results of a type system covering the feature directly. However, resource analysis itself can only use resource equivalent desugaring.

Figure 5.5 compares the JavaScript features covered by the existing systems. Desugaring methods are not considered here, since most of them can be adapted between systems.

5.1.4 Checking versus inference

Type systems can be designed for two different modes of operation. In the type assertion $\Gamma \vdash e : t|\Gamma'$, the expression e is given in both cases. For *type checking* the type t and the type context Γ (and sometimes even Γ') are provided and the purpose of the type system is to validate whether the statement $\Gamma \vdash e : t|\Gamma'$ holds. Statically typed languages like C or Java provide types as part of the syntax of the program. For these languages, type checking is sufficient. A type checking systems for an untyped language like JavaScript requires extra annotation work by the user to provide those types. For the code in Figure 5.1, a valid type checking question would be: "In the context $\Gamma = \{\texttt{x} \mapsto \texttt{Int}\}$, does the object returned by the function body always include the field `a`?" This assertion is formally written as $\Gamma \vdash \mathit{body} : [\texttt{a} : \texttt{String}]|\Gamma'$. Since the typing provided above can be type checked, the answer is "Yes".

On the other hand, the *type inference* problem is to find a t and Γ' (sometimes also Γ) such that the statement $\Gamma \vdash e : t|\Gamma'$ holds for the given expression e. Since type inference is provided with less information, it is more complex than type checking. With less effort for the user of the system, inference is preferable where possible. Initiated by Mitchell [87] and further developed by Aiken and Wimmers [3], many type inference systems reduce the inference problem to a constraint solving problem. For example, in the JavaScript type inference system JS_0^T [9] the constraints are variations of subtyping constraints like $t < [a : (\texttt{Int}, \bullet)]$ which expresses that the type variable t has at least the read-write-able field `a` of type `Int`.

Between type checking and type inference there are various levels of systems which require some annotations to infer the remaining information and infer the remaining information. Some systems require, for example, the headers of all functions to be annotated [21] while the type of all other expressions can be inferred automatically.

A special form of mixed checking and inference is *gradual typing* [109, 106]. It statically infers an incomplete typing from annotations and simple structural constraints. Types which cannot be inferred statically are then dynamically checked during runtime.

5.1.5 Precision

In general, there are multiple possible types for almost all non-trivial expressions. For example, the empty object $[]$ is sound as the type for any object expression, since it assumes nothing about the fields of the object. However, with the type $\texttt{o} : []$ for `o` in

line 6 of Figure 5.1, the expression `o.a + " world"` in line 7 cannot be typed, since the type $[]$ does not allow access to the field a. Therefore, a type system should aim to provide as many sound details as possible.

Formally, consider a type t *more precise* than t' if the two sets

$$T = \{v | v : t\}$$

$$T' = \{v | v : t'\}$$

of all values typed with t or t' are in the subset relation:

$$T \subset T'.$$

Since this defines only a partial order, multiple *most precise* types might exist for any expression e, but in the cases of the systems below, all considered expressions had a unique most precise type.

5.2 Common techniques in JavaScript type systems

The different existing type systems for JavaScript have different goals and use different methods to achieve them. Several common methods are used by multiple system. This section summarises these common methods to present the established state of the art.

5.2.1 Object types

The important data structure in JavaScript are objects. Object types are usually represented as *row types*

$$[field_1 : t_1, ..., field_m : t_m]$$

resembling the object calculus by Abadi and Cardelli [1]. Row types consist of a list of field names with their associated types. For example, the type $[\texttt{a} : \texttt{Int}, \texttt{b} : \texttt{String}, \texttt{c} : []]$ describes an object which has the integer field `a`, the string field `b` and the field `c` which is itself typed as object. To describe nested objects, two methods are common. The first explicitly describes the type of the nested objects like the field `c` in this example. In this case, recursive object types are modelled by an explicit recursive operator: a recursive list with an integer `head` and a `tail` list can be typed as

$$t_{list} = \mu\alpha.[\texttt{head} : \texttt{Int}, \texttt{tail} : \alpha].$$

The recursive operator $\mu\alpha$ describes that every α contained in the type stands itself for the type t_{list}. The type t_{list} is equivalent to its unfolded type $[\texttt{head} : \texttt{Int}, \texttt{tail} : \mu\alpha.[\texttt{head} : \texttt{Int}, \texttt{tail} : \alpha]]$.

Another approach represents the types of nested objects by abstract references l. These references are than mapped to types by the *abstract heap* or *typing heap* X. For example, the abstract location l_{list} could be mapped to

$$X(l_{list}) = [\texttt{head} : \texttt{Int}, \texttt{tail} : l_{list}].$$

With the reflexive inclusion of l_{list} this type describes the same list type as t_{list}. During runtime, each abstract location l may describe multiple concrete heap locations.

The abstract heap simplifies the handling of aliasing. Consider the example in Figure 5.2. Both variables `x` and `y` are lists and they are aliased in line 2. In the

Figure 5.2 Aliasing example

```
1 var x = {head:4};
2 var y = x;
```

explicit approach where both variables are typed as $\Gamma(\texttt{y}) = \Gamma(\texttt{x}) = [\texttt{head} : \texttt{Int}]$ the types are independent of each other and any change of `x` has to be reflected in the type of `y` manually. In the abstract location approach they are both typed with the same abstract location $\Gamma(\texttt{y}) = \Gamma(\texttt{x}) = l$ with the value $X(l) = [\texttt{head} : \texttt{Int}]$ in the abstract heap. A change of the type of `x` changes the abstract heap $X(l)$ which automatically changes the type of `y`.

One weakness of row types is the fact that each field of the object has to be mentioned explicitly. However, JavaScript allows to access arbitrary fields of an object by dynamically computed strings. The expression

```
var end = "il";x["ta" + end]
```

accesses the field `x[tail]` even though `tail` is not mentioned explicitly. If `end` depends on user input, the value of `"ta" + end` is unavailable before execution and cannot be considered in static analysis. In order to overcome this weakness, some systems include a δ-field (similar to the template field [103]) in their row types:

$$[\texttt{head} : \texttt{Int}, \texttt{tail} : l_{list}, \delta : []]$$

This expresses that all fields not mentioned explicitly can be typed as $[]$. Since the δ-field potentially describes a set of very different fields it has to over-approximate and

is less precise than the explicitly mentioned fields. Guha et al. [48] take this idea one step further by allowing patterns in the object field names to describe multiple fields with each entrance.

5.2.2 Singleton and summary types

JavaScript's *strong updates* also pose a significant challenge to type systems. With a strong update the type of a variable can be changed, simply by assigning a different value. This means each assignment can potentially change the type structure of the assigned location. In JavaScript strong updates are common, since assignment to a fresh field $o.m$ of the object value o implicitly extends the type of o to contain the field m. Static type analysis typically describes multiple concrete values by the same type variable. Consequently, a strong update cannot update the type without generating false information for other values. Consider the example in Figure 5.3, which constructs an array of objects in a loop.

Figure 5.3 Loop example

```
1 var i=0;
2 var x;
3 while (i<5)
4   x[i] = {};
5 x[4].a = 3;
```

After the loop in line 5, the objects `x[0]` to `x[4]` are typically typed with the same type variable, since they were created by the same expression. Therefore, strongly updating the type of `x[4]` to contain the new field `a` with value 3 changes the type of all those objects. This situation can be handled by a system involving singleton and summary types [64, 52]. A *summary type* describes multiple values at the same time and cannot be strongly updated. In contrast, a value can be typed with a *singleton type* which describes only this specific value and allows strong updates. When the singleton property of a type cannot be guaranteed anymore, e.g. when the variable is used as parameter to a function or the constructor used for this variable is invoked again, the singleton type has to be converted into a summary type. In this example, the type for `x[4]` is modelled by the singleton type $[]$ to allow the strong update in line 5. The types for `x[0]` to `x[3]` are described by one summary type $t_{array} = []$.

Singleton types and summary types can refer to each other. For example a singleton object can refer to a summary type as a field type. In the example above, replacing the

last line by `x[4].last = x[3]`, the type of $x[4]$ would be a singleton type of the form $t_4 = [\texttt{last} : t_{array}]$ which includes the summary type t_{array}.

The singleton / summary paradigm supports in particular the initialisation pattern: it is a common assumption in analysis systems for JavaScript that the structure of objects is mostly changed using extensions and deletions, directly after the creation of the object, thought of as an *initialisation* phase. Afterwards, the structure of the object can be assumed fixed. Therefore, an object can be typed with a singleton type until the initialisation is completed and then be joined into the appropriate summary type. However, this pattern was evaluated [101] on a set of benchmarks with the conclusion that the initialisation phase is not easily identifiable in real-world JavaScript code.

5.2.3 Function types

The ECMA standard [61] stores a function value as JavaScript object with the property `__proto__` set to the special build-in object `Function.prototype`. Apart from this property, the function object contains a few more internal properties to store the function's code, it's expected parameters and scope. After a function `f` has been defined, an expression like `f.m=4` can extend the function object with additional fields as any other object. This feature is, for example, used to present the API of the popular jQuery library in a concise way.

Some type systems for JavaScript imitate this specification by typing functions with object types. Other systems define separate function types to simplify the analysis of function executions. This simplification, however, cannot type fields of functions directly. In this case, the function object has to be desugared into a pure function and an associated object which stores the extended fields.

Inside JavaScript function bodies, the variable `this` refers to the *receiver* of the function call. The receiver of an object method `o.m()` is the owning object `o`, the receiver for calls as constructor `new f()` is a new empty object and for normal function invocations `f()` the current global scope object acts as the receiver. The variable `this` storing the receiver is an implicit parameter and the function type has to reflect its expected type. Most systems explicitly state the expected receiver type O in the function type as $O \times t_x \to t_{ret}$.

Since the behaviour of a function call differs depending on whether it is called as a method, constructor or simple function, some systems choose to differentiate between the three different kinds of function calls in the syntax or in the types. For exam-

ple, Zhao [120] restricts the names of constructors to start with a capital letter, while methods and simple functions start with a lower-case letter. Function in real-world JavaScript code can be used in either role.

5.3 Typed JavaScript derivatives

The survey of type systems below only considers type systems for pure JavaScript. In addition, there are a few derivations of the JavaScript Language which include types. TypeScript [85] developed by Microsoft adds types to a language based on JavaScript. It extends the syntax of the language by adding type annotations into function definitions and other statements and provides additional features like classes and class interfaces which are compiled into common JavaScript programming patterns. TypeScript in Version 1.8 can even emit code in ECMAScript 6 strict mode. During compilation TypeScript checks all available (provided and easily inferable) types and issues warnings for type mismatches. The resulting typing is not complete, meaning it allows untyped values as long as they are not violating other type constraints. All type information is eliminated during compilation to create compatible JavaScript code.

Google's alternative, AtScript [45], is a system extending TypeScript, besides other features, with generated runtime type assertions which dynamically check the types of typed values during runtime where static checking is not possible. Like TypeScript, the resulting types are not complete and its goals value flexibility and usefulness over soundness. In 2015 AtScript was announced defunct and some features merged into TypeScript.

GoriallaScript [32] is another extension of JavaScript, which checks available type information during compilation into JavaScript. It restricts the polymorphic operators like the + operator to type-safe behaviour. In addition to the basic JavaScript types GorillaScript allows to define finite types by arrays containing all possible values of the new type. A value can then be dynamically checked during runtime against the defined types, similar to the `typeof` operator in JavaScript

The language asm.js [89] is a subset of JavaScript which, when compiled by so called ahead-of-time compilers, provides guarantees for the runtime values, can eliminate JavaScript's dynamic type checks and optimises the memory behaviour. The goal is to imitate behaviour of compiled languages as C and improve the performance of JavaScript programs.

5.4 A survey of type systems for JavaScript

This section discusses the details of existing type systems. The abstract evaluation systems TAJS [64] and JSAI [69] are also considered, since their abstract domain is reasonably close to the structure of object types. Other systems (e.g. separation logic) can provide similar information as output, but the translation into the desired type structure would require a considerable amount of work. Those systems are instead mentioned in Chapter 2.3.2.

The results of the survey are summarised in Figures 5.4 and 5.5. Figure 5.4a shows which systems built upon each other considering reuse of semantics, type syntax or implementation. The properties which are interesting for extending a system with AmorJiSe are summarised in Figure 5.4b and Figure 5.5 summarises the subset of the syntax covered by the systems directly.

The optimal candidate to build AmorJiSe upon is a formally proven, sound system for type inference with an implementation.

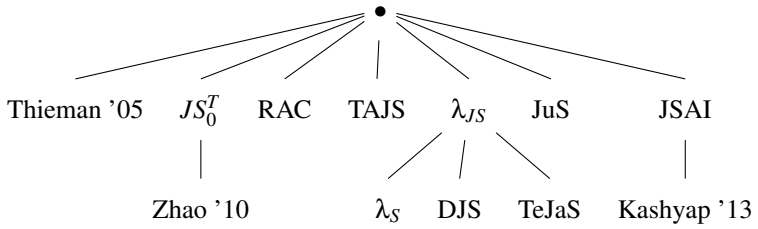
The three candidates fulfilling all requirements are JS_0^T, its extension by Zhao and RAC. RAC analyses a lambda calculus with JavaScript-similar object behaviour. JS_0^T instead analyses JavaScript directly and specifically represent object extensions in a simple and controllable way. Since object extension might require additional heap resources, this explicit presentation is beneficial during resource analysis. Consequently, the amortised system AmorJiSe is built closely resembling JS_0^T. The TeJaS system has a similar notation and properties. However the soundness of such a modular systems is an open problem. A proven instance of this system could also be used as underlying system. The analysis of implicit type checks, as performed by Kashyap [71], does not interfere with AmorJiSe's analysis and could be used to increase the precision of the chosen underlying system further.

The following sections discuss the details of the existing systems. For each, it will briefly describe the methods, the covered part of the language, the soundness result, the required annotations and the state of the implementation.

5.4.1 Type system by Thiemann ('05)

The first attempt at a type system [114] by Thieman presents a static type system for a core of JavaScript with the goal to highlight potentially unintended type coercion. Typed programs are guaranteed to be free from calls to non-function values, arithmetic operations applied to non-numeric values, reading from non-existing variables and ac-

Figure 5.4 Existing type systems for JavaScript



(a) Relationships

	Soundness proof	Checking / inference	Coverage	Implementation
Thiemann '05 [114]	pen & paper proof omitted	checking	core	not available
JS_0^T [9, 7, 8]	pen & paper incorrect	inference	core	used in evaluation
Zhao '10 [120, 119]	pen & paper	inference	core	tested on small programs
RAC [52, 115]	pen & paper	inference	core	source available
TAJS [10, 64, 65]	sketched	inference	full	source available tested on benchmarks
λ_{JS} [49]	sketched	checking	desugar	source available tested on benchmarks
λ_S [50]	pen & paper	checking	desugar	source available
DJS [21]	not provided	partial inference	desugar	source available tested on benchmarks
TeJaS [74]	Pot. unsound pen & paper	checking (general) inference (Instances)	desugar	source available tested on real-world code
JuS [41]	not provided	partial inference	core	source available tested on benchmarks
JSAI [69]	not provided	inference	desugar/full	source available tested on benchmarks
Kashyap '13 [71]	ref. JSAI	inference	ref. JSAI	source available tested on benchmarks and real-world code

(b) Properties

cessed fields of `null` values.

The object types are row types with δ-fields. Additionally, the object types can express wrappers around simple types like integers or strings. This way, the result of the expression `x=4;x.m=5` can be typed as $Object(\texttt{Int})[\texttt{m}:\texttt{Int}]$ to describe a wrapper object around an integer which has been extended with the field `m`. Function types are handled using a dedicated type $Function(this:\tau;\rho\to\tau')$. The parameters ρ are modelled as a row type which allows for variadicity. To increase the precision, sum types $\texttt{Int}+[]$ describe values which are either `Int` or an object.

The formal JavaScript subset is given as small-step operational semantics. It considers functions including the handling of constructors and methods, but omits function

Figure 5.5 Candidates for underlying system - JavaScript Syntax coverage

		Objects							Functions													
	nested expr.	{}	[]	.	+[1]	-[2]	protot.	Arrays	Stmt	expr	()	new	o.m()	var	if	for	while	with	break	errors	eval	scope	libs
Th05	✓	✓	✓	x	✓	×	×	×	×	✓	✓	✓	✓	✓[4]	✓	×	✓	×	×	×	×	×	×
JS_0^T	✓	×	×	✓	✓	×	×	×	✓[3]	×	✓	✓	✓	×	✓	×	×	×	×	×	×	×	×
Zhao	×	×	×	✓	✓	×	×	×	✓[3]	×	✓	✓	✓	×	✓	×	×	×	×	×	×	×	×
RAC	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×[5]	×	×	×	×	×	×	×	×	×
TAJS	full language																						DOM jQuery
λ_{JS}	✓	✓	✓	×	✓	✓	×	×	×	✓	✓	×	×	×[5]	✓	×	✓	×	✓	✓	×	×	×
λ_S	✓	×	×	×	×	×	×	×	×	✓	✓	×	×	×[5]	✓	×	×	×	✓	×	×	×	×
DJS	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓	×	✓	×	×	×	×
TeJaS	see λ_{JS}																						
JuS	✓	✓	×	✓	✓	✓	✓	×	×	✓	✓	✓	×	✓	✓	×	✓	✓	×	×	×	✓	×
JSAI	✓	×	✓	×	✓	✓	×	×	×	✓	✓	✓	✓	×	✓	✓	✓	×	✓	✓	×	×	DOM
Ka13	see JSAI																						

[1] Object extension by assignment
[2] `delete` operator
[3] JS_0^T and Zhao only allow function statements at the top level
[4] Th05 only allows `var` statements at the top of a scope
[5] RAC, λ_{JS} and λ_S use let normal form

statements in favour of function expressions and requires local variables to be defined at the beginning of the function body. The object operations are modelled including object literals and read, write and extend operations for objects in the form of computed access $e_1[e_2]$. Static access $e_1.m$ is not handled, but instead identified as equivalent to the computed access $e_1["m"]$. Considering statements, the core language includes while-loops and if clauses.

The main result of [114] is the type soundness which is proven via reduction and progress. The implementation for the type checking algorithm was stated to be on the way, but to the best of my knowledge, has not been published. The author has since contributed to other type systems considered below [52, 65, 64, 115]. Type inference is not considered in this system.

This system handles many important features of the JavaScript language and employs a promising method to express simple types coerced into objects, which is unique to this system. The formally proven soundness makes this system interesting to extend, but as underlying system for AmorJiSe the missing inference would lessen the advantage of the automatic inference of the amortised annotations.

5.4.2 JS_0^T

Anderson and Drossopoulou developed the system JS_0^T [8, 9]. Its goal is to prove the absence of runtime errors for typed expressions.

JS_0^T uses recursive row types for objects and adds the markers $\circ/\bullet$ to the type of each field. The marker definite $\bullet$ describes that this field is guaranteed to exist in this object with the given type, whereas the potential marker $\circ$ asserts that the field will have the given type if it gets added to the object in the future. Hence, potential fields can be written to but not read. Using those markers, JS_0^T is able to express a controlled subset of strong updates.

The language covered includes only the variable `x` in addition to `this` and allows read operations and weak updates for variables (except for strong object extensions). The authors claim that additional variables can be simulated, but the simulation is not as straight forward as claimed (see the remark in Section 7.1.6). Furthermore, JS_0^T supports objects with read, write and implicit extension operations, function expressions, calls to functions, object methods and constructors and a conditional expression. Other features like loops, local variable definitions, function statements and object literals are omitted. The formal definition of this language is given as big-step operational seman-

tics.

The proof for the type soundness is given, but is unfortunately not correct, as shown in Chapter 7 along with a fix. A type inference algorithm is provided and partly proven sound in relation to a big-step operational semantics. The inference is realised by generating subtyping constraints. From the transitive closure of those subtyping constraints the solution can be extracted efficiently. The implementation of the inference algorithm extends the formal core language with a few more features of JavaScript without a formal soundness proof. As evaluation the implementation was successfully tested on a few crafted example programs.

With a formally proven soundness theorem (given the proposed fix) and inference this system is a good candidate as underlying system for AmorJiSe. The controlled handling of strong updates provides exactly the right information needed for the object model. For this reason, AmorJiSe in the following chapter is modelled after JS_0^T.

5.4.2.1 Type system by Zhao ('10)

In two papers [119, 120], Zhao presents a system, which builds upon JS_0^T. The purpose of this system is to infer the structure of objects. In comparison to JS_0^T, this system can also track which object fields are removed using JavaScript's `delete` operator.

In addition to the $\circ/\bullet$ mechanics of JS_0^T this system uses singleton / summary object types. The added set of strings M in the function type $(O,M),t \to t'$ describes the fields that are added to the type O during the execution of the function body. This increases the precision of the object types in comparison to JS_0^T. In the follow-up paper [120] object types are instead represented as t/C where t is a type variable and C is a set of constraints on the type variables. The list type could for example be described as

$$t_{list}/\{t_{list}(\texttt{head}) \leq \texttt{String}, t_{list}(\texttt{tail}) \leq t_{list}\}$$

where t_{list} is a type variable. The constraint sets C are simplified and checked for consistency after each modification to decrease the size and complexity. For each function call the system freshly instantiates the type variables in function types to handle the functions polymorphically without analysing the function body multiple times. Only mutually recursive functions have to be analysed strictly monomorphically.

The formal language of Zhao covers slightly less than JS_0^T. Nested expressions are replaced by assignments to local variables and the syntax of constructors is artificially differentiated from object methods. Normal function calls are excluded in favour of

method calls. The language allows for the definition of local variables without hoisting.

The inference algorithm and the soundness proof are analogue to JS_0^T. Since this work only considers the type inference algorithm it avoids the flaw in the type checking of JS_0^T. A prototype implementation is available.

Some extensions made in this system can be used to extend JS_0^T as the underlying system for AmorJiSe. However, the types derived by this system are not explicit but rather expressed as type variables with constraints. To validate a property of a type, the authors add the property to the constraint set of the type variable and verify the consistency of the constraints set. This method is not efficient in the context of AmorJiSe, since the system often has to read the type of a field from an object.

5.4.3 RAC

The system RAC [115, 52] stands for Recency-Aware Calculus and is defined by Thiemann and Heidegger. RAC is a type system for a lambda calculus with objects which guarantees that a typed expression will not result in a runtime error.

The recency types implement the singleton / summary type paradigm in the way that the most recent object created by a constructor expression is always typed as a singleton type. Object types use abstract locations in a typing heap, which is split into a singleton heap and a summary heap. The static analysis assumes the code contains the explicit demotion operator $\natural$ to merge a singleton type into the corresponding summery type before the same constructor is used again. If, for example, a new object is created using the `new` operator at the code location l_1

```
1  let f=λx.(new^l1) in
2    f(42)
```

then the singleton type stored at the abstract singleton location l_1 has to be explicitly demoted and merged into the summary type at this location before the function `f` is called.

```
1  let f=λx.(new^l1) in
2    ♮^l1; f(42)
```

In a pre-processing step the system automatically injects all needed demotion operators into the code.

Function types

$$\texttt{f} : (\Sigma, t) \xrightarrow{L} (\Sigma', t')$$

track the side-effects on the types of the outside scope. Σ documents the requirements on the singleton heap of the call side and Σ' summarises the effects on the singleton heap after execution of `f`. The set L specifies which abstract locations need to be demoted before the call. In the above example `f` is typed as $([], \texttt{Int}) \xrightarrow{\{l_1\}} ([l_1 \mapsto [\,]], obj(@l_1))$ to force demotion of the singleton location l_1 before execution and to ensure this location is typed as empty object after execution. This function does not have requirements on the call-site singleton heap.

The similarities of the analysed call-by-value lambda calculus to JavaScript arise from the inclusion of first-class functions, prototypal inheritance and objects with read, write operations and the explicit extension of objects. Computed read and write operations $e_1[e_2]$ are not handled. In contrast to JavaScript, RAC relies on let bindings instead of variable assignments, does not analyse nested expressions and does not have constructor functions. Instead, it uses an universal object constructor `new` equivalently to the empty object literal $\{\}$ in JavaScript. A transformation of the missing features is not provided. The language is defined by small-step operational semantic rules, which also uses an explicit singleton and summary heap to reflect the split in the type heap. Each singleton type location is mapped to exactly one concrete singleton heap location, while summary type locations are mapped to a set of concrete locations in the summary heap.

The soundness of the type system is formally proven in the technical report. A type inference algorithm and an implementation is provided based on constraints on upper and lower type bounds.

As underlying type system for AmorJiSe, the language considered in this system is too far from JavaScript. However, the recency paradigm as implemented here could increase the precision of the system JS_0^T.

5.4.4 TAJS

The system TAJS by Thiemann, Andreasen, Møller, Jensen [64, 65, 62, 10] presents an abstract interpretation based system which infers the structure of runtime values for JavaScript.

TAJS builds a control flow graph consisting of nodes for each subexpression and edges for control flow between the expressions. TAJS, furthermore, defines abstract states which over-approximate the runtime state at each node in the graph. To describe the state modification of each node, transition functions are defined describing the rela-

tion between the abstract state before and after execution for each JavaScript construct. The analysis computes the fixpoint of all transition function across the graph.

Functions are handled context sensitively by computing a fixpoint of the function body separately for each function call. TAJS is further extended with determinacy analysis [10], such that the values of constant expressions are propagated through the graph. This information is used to decide on loop unrolling and thus increase precision.

TAJS introduces a method called `unevalize` to analyse dynamic code execution by JavaScript's `eval` and similar. It takes common programming patterns using `eval` and translates them into equivalent code using for example the method `JSON.parse` which deserialises an JavaScript object from code without side-effects.

The required transition functions are presented for all commands of the bytecode language used in real-world interpreters and includes handling of functions, variables and objects including read, possibly extending write and delete operations. As statements, TAJS considers function invocation, including constructor calls, `if`-clauses and error handling. In addition, the transition functions are provided for a collection of built-in functions like the browser specific functions and the DOM API.

In earlier versions of TAJS the authors argue informally that the reachable state space is limited enough to guarantee termination of the fixpoint iteration. The evaluation of a prototype implementation of TAJS returns sound results in a manageable execution time. In the later version the fixpoint computation is restricted to 1000 iterations. If the fixpoint is not reached by this point, the current abstract state is used as an potentially unsound approximation of the fixpoint. Unfortunately, additional soundness bugs in connection with the handling of `try-catch-finally` clauses were found in TAJS [69] which are attributed to the lack of formal specification of the handled language.

The large language covered by this system makes it interesting as underlying system, especially the unevalize translation. Functions are analysed in a context-sensitive way, including variadicity and polymorphism, and the DOM library is vital to analyse mobile apps. However, the only informal soundness argumentation and the potentially unsound approximation are a disadvantage. Using this system as the underlying system for AmorJiSe would also mean that AmorJiSe has to analyse the same intermediate representation, which makes it more difficult to recognise the API calls.

5.4.5 λ_{JS}

The system λ_{JS} by Guha, Saftoiu and Krishnamurthi [49] defines big-step semantic rules for a core JavaScript. An example type system certifies the absence of communication via the function `XMLHttpRequest`. While this type system does not describe the necessary information about the data structure of values, the semantics of λ_{JS} is the basis for multiple other systems.

The core system's features are chosen to cover full JavaScript language via a desugar function. It includes the object operations for read, write, extend and `delete`. Prototypes and errors are handled and a simplified model of the scope chain is included. The formal language handles references, but in contrast to JavaScript they are created and dereferenced using explicit operators. λ_{JS} includes simple function expressions, which are used to desugar function statements, object methods and constructors as function objects similar to the ECMA standard. Considering statements, λ_{JS} includes `while` loops, `if` clauses and `break` with labels. All other language constructs are translated into the core calculus. The authors claim to even desugar the non-static scope behaviour into the lexically scoped core language by making the lookup procedure explicit. Unfortunately, the translation increases the size of expressions by an average factor of 200.

The semantics itself is proven sound with the result that well-formed expressions do not get stuck but evaluate either to a value or to an error. The completeness of the desugar process is evaluated by translating a set of real-world benchmarks into the core language and the implementation is available for download.

In conclusion, the language λ_{JS} is suitable for AmorJiSe, even though the resource effect of the desugar method would impose additional work. However, this work does not provide a suitable type system.

5.4.5.1 λ_S

The calculus λ_S [50] by Guha, Saftoiu and Krishnamurthi translates JavaScript into a core language inspired by λ_{JS}. The type system defined on this core language certifies the absence of runtime errors.

λ_S employs so called tags which resemble dynamic types. During runtime, the tag of a value can be validated using a `tagcheck` which results in a `tagerr` if the tag does not apply to the value. The analysis uses a flow analysis to insert non-failing `tagcheck`s into the source code. This additional information obtained by these checks

can than be used to infer precise types for the code. The system analyses each function body in isolation to make the system scalable. This method comes with the loss of inter-procedural precision. The types incorporate reference types and abstract heaps to handle aliasing and strong updates correctly even through function calls. Types can be combined into union types to increase the precision and coverage. The function types ignore the receiver `this` and handle method and constructor invocations with receiver by desugaring.

The core language λ_S includes function expressions and function calls, references with an explicit reference and dereference operator, `if` clauses and the `break` statement with labels. The translation from JavaScript into the core language is not provided in the paper but the implementation is available.

The core system is formally proven correct via progress and preservation. The location of `tagcheck`s can be inferred automatically, but the inference of the types is not considered. An implementation based on a monotone framework [73] is provided and example programs have been analysed within a few seconds.

Since λ_S is proven sound, especially its reference types could be used in an data type system underlying AmorJiSe. However, type inference is missing and the covered language is less than covered in the system by Thiemann ('05).

5.4.5.2 DJS

The system DJS [21] introduced by Chugh, Herman and Jhala presents dependent refinement types for JavaScript. The authors adjust their system D to JavaScript to obtain the language D!. On this language the type system DJS is defined.

The main characteristics of DJS is the refinement notation. A type $\{v \mid p(v)\}$ is defined as the set of all values v satisfying the property p. Syntactic sugar is provided to express simple types like `String` or `Int`, but more expressive types can be specified, for example, the type *falsy* combining all values which evaluate to `false` when coerced to Boolean. The property p can depend on other values in the current abstract heap which makes this notation powerful.

In order to track prototype relations without state explosion, the heap is modelled as a shallow and a deep heap. The shallow heap stores precise information about the prototype chains of crucial types. Less crucial types can be estimated in the deep heap which only summarises all fields contained in the prototype chain without storing the whole chain itself. This keeps the complexity of long prototype chains manageable. DJS also uses singleton / summary types to handle strong updates. Summary types can

be thawed and frozen to allow temporary modifications of their structure. Functions are modelled with a dedicated function type and aliasing is handled by reference types.

The considered core language D! is based on λ_{JS} and includes prototypes, strong updates, higher-order functions, arrays, Boolean operations, but prohibits implicit coercion. A purely syntactical desugar process is provided which translates the uncovered language features into D!. DJS is even able to analyse some common patterns using `eval` by translating them into controllable JavaScript features such as the JSON library.

A type checking algorithm generates subtyping relations from given types and uses an SMT solver to validate them. The type checking is only proven sound by reference to the soundness proof for System D. Types for local expressions can mostly be inferred, but type annotations for function types are required including types for the accessed part of the scope chain. The annotation overhead on example programs constitutes about 70%. The provided implementation, which was used for experiments, is based on the parser for λ_{JS}.

The advantage of this system as underlying system for AmorJiSe is the handling of the scope chain. The needed annotations and missing formal soundness proof make it less useful for this purpose.

5.4.5.3 TeJaS

Lerner et al. present the system TeJaS [74]. Rather than a type system on its own, it is a framework to implement type systems for JavaScript. It is distilled from the similarities between various other specialised systems and the semantics is adapted from λ_{JS}. The focus of the framework lies on the modularity: Different parts can be implemented independently and combined to construct different systems.

The authors identify the 7 essential parts of a type system: type and kind syntax, environments, kind checker, type checker, subtyping, decorator (weaving types into expressions) and the desugar method. TeJaS provides a default implementation for each part which can be extended or redefined by each type system instance. The default types include basic types like `Int`, `String` (modelled as regular expressions), objects (abstract references into a typing heap with prototypes), functions (modelled as objects) as well as unions and intersections of these basic types. Like JS_0^T, this system defines a state for each object field, but TeJaS is more expressive than JS_0^T. Apart from the definite ! (equivalent to $\bullet$ in JS_0^T) and potential ? (equivalent to $\circ$ in JS_0^T), the value `Absent` expresses that the field is definitely not present in the object directly,

^ expresses that the field is present somewhere in the prototype chain and the value `Hidden` prevents access completely.

The type system in TeJaS is defined relative to λ_{JS}. Therefore, it covers the same language with objects, including prototypes, function expressions, error handling and the limited control flow operations, including `if` clauses and labelled `break` statements.

With regards to the soundness of the base system, the authors argue that the soundness proof of the basic system with the default implementations should be equivalently to the systems it was distilled from. However, they admit that the soundness of the modular system, even when assuming soundness of all modules, is an open problem. Instead, they mention the use even of potentially unsound modules as opportunity to capture the dynamic features of JavaScript better. The paper discusses several example type systems implemented with the framework which have been proven sound in previous work.

Type inference for the system in general is proven undecidable. But in the discussed instances inference is possible in reasonable time. Type checking uses all provided annotations bidirectionally to benefit from the combined power of inference and type checking. Syntactic sugar for the annotations is presented to represent common programming patterns concisely and a dynamic companion analysis estimates type annotations to reduce the annotation work. An implementation of the framework is provided along with the implemented instances of type systems using the framework.

This system uses a similar object extension modelling as JS_0^T, but additionally models prototype links. A custom type system instantiated from TeJaS sounds promising as more expressive underlying type system than JS_0^T. However, the soundness theorem would need to be proven formally.

5.4.6 JuS

Gardner, Naudziuniene and Smith introduce the symbolic execution system JuS (JavaScript under scrutiny) [41]. It propagates logic formulas through abstract states. The system aims to produce corner cases of the program which invalidate a specified property.

The program logic used in JuS is adopted from the separation logic introduced by Gardner et al. [42]. The logic statements cluster information into so called *StoreLet* which roughly describes a part of the scope chain with the information about the vari-

ables and fields which are definitely not defined in the scope, maybe defined or definitely assigned a specific value. Nested and recursive objects are stored as references to handle aliasing correctly. The StoreLet abstraction contains information about the object structure and can be translated into object types with little effort.

The language covered contains String, numeric and Boolean constants, object literals, read and write access to objects and simple functions which can be called as constructors or methods. Function statements are not defined. Concerning statements, JuS includes the `if` clause, `while` loops and the `with` block.

JuS aims for soundness and the results are meant as a guarantee of the proven property. However, the soundness theorem is not formally specified and neither is a proof. The system uses a mix between checking and inference to infer as much information as possible, but loop invariants need to be inserted as annotations in any case. An implementation is provided online as part of the JSCert project.

JuS presents an interesting extension of the scope handling, which AmorJiSe could take advantage of. However, to be used with AmorJiSe directly, the needed annotations and the missing formal soundness theorem are a disadvantage.

5.4.7 JSAI

Kashyap et al. present the system JSAI [69, 70] which aims to provide a formal framework to implement static analysis for real-world JavaScript code. The focus of JSAI is on configurability: the abstract domain for values is implemented as module which can be swapped easily and the merge operator to combine several concrete states into one abstract state can be defined separately. This enables JSAI to implement a range of sensitivities like context-sensitivity, flow-sensitivity, object-sensitivity and their k-bounded finite versions. In total, the papers evaluate 56 different strategies for sensitivity themselves.

The static analysis is based on an abstract interpretation state machine, which combines different methods for the different parts of the abstract state. They employ type-analysis, pointer analysis, string analysis, constant propagation and control-flow analysis in a manner that each analysis can benefit from the result of the other. The used object types consists of 2 row types. The first row stores ordinary object fields while the second row tracks the types of class-specific fields, for example, the `length` property of arrays and the local scope object of functions. This way, JSAI can analyse ambivalent operations of objects modelling more complex classes with more precision.

The language covered by JSAI is an intermediate language called notJS which is not designed as a pure core language but implements more complex features directly. The execution of notJS is implemented and checked against benchmarks comparable to evaluations of other real-world JavaScript implementations. The intermediate language includes basic values as well as objects with field read, write, extend and deletion, function expressions, calls and constructors as well as `while` and `for` loops, conditionals and `jump` to model further control-flow. Error-handling is also included. Functions and arrays are handled as objects in correlation to the JavaScript standard. In an effort to transfer their results to real-world JavaScript, the authors present a translation from JavaScript into nonJS.

The whole system is modelled formally and annotations in the code are not required. However, neither the soundness property nor the proof are provided in the paper. The same is true for the translation from JavaScript into notJS. The implementation and further material is available.

As underlying system for AmorJiSe, JSAI has many advantages. It handles real-world JavaScript with many different analysis methods and handles more language constructs directly. Like in the case of TeJaS, a custom instance of this system with a formal model would be a good candidate to increase the precision and coverage of AmorJiSe. The resource equivalence of the translation between JavaScript and notJS has to be proven separately.

5.4.7.1 Type system by Kashyap et al. ('13)

Kashyap et al. use JSAI to implement a refinement based system [71] to increase precision of static analyses using the information obtained from type checks contained in the JavaScript syntax.

In addition to obvious type checks using the JavaScript operator `typeof`, this work identifies several implicit type checks in JavaScript programs. An accessed object is neither `null` nor `undefined`, only functions can be called and a value which is converted by JavaScript has to be primitive (neither object nor function).

The types used in this system are primitive types (*num,bool,str,null,undefined*), object types and function-objects. Those basic types can be combined as union types. The type $\texttt{null} \cup \mathit{ObjTypes}$, for example, specifies values which are either `null` or an object. From explicit and implicit checks contained in a program, the system derives conditions on the type variables. For example, $\texttt{isPrim}(t)$ describes that t is one of the primitive types and the constraint $\texttt{typeof}(t) = \texttt{"function"}$ specifies the type t as a

function type.

The abstract states of this system are kept very simple with the purpose of demonstrating the power of the implicit constraints rather than a complicated type system. As such it rather meant as an addition to an already existing type system than a type system by itself. The covered language is not specified explicitly but the examples include operations on objects including read, write, extend and delete, `if` clauses, higher-order functions as methods and simple functions. Like JSAI, this extension does not handle `eval` and similar dynamic JavaScript features.

The authors claim soundness for the system and argue informally that all necessary conditions are fulfilled. The formal soundness proof or the formal soundness theorem are not included. The implementation uses the JSAI framework to infer the abstract values without code annotations and has been used on standard benchmark suits, real-world applications from open-source projects and machine generated code.

The presented method in this system is meant as an addition to an existing type system. It could also be added to increase the precision of the AmorJiSe system. To be considered as an underlying type system itself, the considered language and soundness would have to be specified formally.

Chapter 6

The system AmorJiSe

This chapter introduces the system AmorJiSe which extends types for JavaScript with amortised annotations. While the original types describe the structure of the values, the amortised annotations describe resource units reserved to be used in computations with the typed values. AmorJiSe automatically infers those annotations and this way describes the maximal resource consumption of the typed expression.[1]

The following sections show the basic techniques of amortised systems and present AmorJiSe with its types, its properties, an inference algorithm and the annotated semantics for JavaScript which models the resource consumption of JavaScript expressions. The analysed language covers the subset of JavaScript which manipulates objects and is later extended with the handling of function expressions.

The main result of this chapter is the proof of soundness for the amortised type system. It guarantees that the resource consumption of the analysed expression, according to the annotated evaluation, is less than the derived bound.

6.1 Terminology

AmorJiSe is a *type system* which derives a bound on the maximal resource usage of the *analysed code*. The *language constructs* and *critical* APIs of an expression are assumed to consume *resource units* according to the *resource model*. Each resource access is accounted for from the number of available resource units.

AmorJiSe requires *data types* for each expression as input, which describe the structure of the evaluated value of the typed expression. The derived *amortised an-*

[1] A preliminary version of this system has been published at the International Symposium for Symbolic Computation in Software Science (SCSS, 2014) [39].

notations inside the *amortised types* symbolise a number of *reserved* resource units. Reserved units are already accounted for in the overall resource consumption, but they can be used in a later stage to call a resource consuming API. Units reserved in a typed value are *associated* with this value. Amortised annotations in the function type and the types of the parameters of functions describe the resource units *required* to execute the function.

Within the following definitions and discussions all amortised annotations are displayed in *blue* to differentiate them from the underlying type system and its judgements.

6.2 Overview

Amortised analysis tracks the number of resource units used or reserved during execution of a statement or expression. Each resource access needs to be accounted for either from the supply of globally available resource units or from the resources previously reserved. Reserved resource units are similar to the tickets in the PhoneWrap system, but can be stored alongside the data structures of values to describe data-dependent resource bounds.

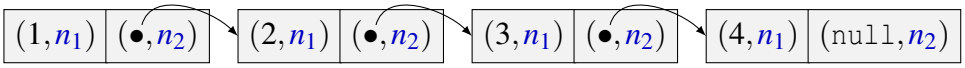
To describe the global resource requirement of an expression, the amortised type judgement states the number of resource units globally available before and after the evaluation of an expression. The judgement

$$\Gamma, n \vdash e : t | \Gamma', n'$$

states that in the context Γ the expression e can be executed with n available resource units and after execution n' resource units remain available. For example, if an expression e consumes 2 resource units and is executed with $n = 5$ available resource units, after the evaluation $n' = 3$ resource units remain available.

With these annotations in the type judgement the system can express constant bounds. As discussed previously, this does not cover the behaviour of modern JavaScript apps as the resource consumption generally highly depends on the size of variables and inputs. The power of amortised types comes from the injection of amortised annotations into recursive data types. Each amortised type $t^+ = (t, n)$ is a pair of a data type t and a number of reserved resource units n stored with a value of this type. In recursive types, like the list type $t_{list} = \mu\alpha.[\texttt{head} : (\texttt{Int}, n_1), \texttt{tail} : (\alpha, n_2)]$, the annotations n_1 and n_2 state the number of reserved resource units for each element in the

recursive structure. This results in a total number of reserved units dependent on the size of the data structure. For example, a list of type t_{list} with length 4



stores $4 \cdot (n_1 + n_2)$ reserved resource units in total. By using such a type as the parameter for a function, AmorJiSe can describe the resource consumption of the function dependent on the size of the input without using dependent types. After inferring types for all functions of a program, similar bounds can be derived from the result as enforced by the dynamic analysis PhoneWrap.

During type checking $\Gamma, n \vdash e : t | \Gamma', n'$ the typing context Γ stores annotated types for all variables. Therefore, the total number of resource units available to an expression e is the number of globally available resource units n in addition to the sum of all reserved resource units stored in the variable types in Γ. This total number of resource units available is called the *potential* of the typing state (Γ, n). Since the amortised annotations in recursive types describe the number of units per element, the total potential also depends on the values of the variables during runtime. Even though those runtime values are not available during analysis, the soundness property of amortised analysis ensures that during analysis the potential is correctly decreased for each resource access and never increased, independent of the actual runtime value.

The basic information used by AmorJiSe to determine the resource consumption of an expression is provided by the resource model, which describes the resource consumption of language constructs and API functions. The resource consumption of each language construct, e.g. of each function calls or each object extensions, are each modelled by a constant. The resource behaviour of the API functions are described by their function type in the initial typing context Γ_0. This way, referring to Chapter 2.2, AmorJiSe can analyse API activated and language activated resources.

So far, amortised types have mostly been studied for functional languages. JavaScript is an imperative language and its *side effects* present an extra challenge to amortised types. In order to provide an overview of the challenges faced while implementing amortised types for JavaScript, consider the uploaded pictures as resource in the fictitious app “ Pick-Up” with the following code examples. The API function `UPLOAD` serves as an example for a resource consuming function.

Example 6.2.1.

```
1 function upload_list(list) {
2   if (list != null) {
3     UPLOAD(list.value);
4     upload_list(list.next);
5   }
6   return 0;
7 }
```

The function `upload_list` takes a linked list of pictures as input and uploads each picture individually. Since `upload_list` is recursively called for each element in the list, the resource consumption of the original call depends on the length of the input list. Assume the underlying data type for the parameter `list` is the type

$$t_{\texttt{list}} : \mu\alpha[\texttt{value} : \texttt{String}, \texttt{next} : \alpha].$$

It describes that the linked list is implemented as an object with the fields `value` storing the location of the picture as string and `next` which recursively stores the list's continuation of the same list type. According to this, AmorJiSe derives an amortised type

$$t^{+}_{\texttt{list}} : \mu\alpha.[\texttt{value} : (\texttt{String}, 1/1), \texttt{next} : (\alpha, 0/0)].$$

The amortised annotations R/W determine that R resource units are reserved for a field. The second annotation W indicates how many resource units a value has to provide to be assigned to this field. Here, the field `value` stores 1 resource unit and each value assigned to `value` has to provide 1 resource unit. The full type for the function `upload_list`

$$t^{+}_{\texttt{upload\_list}} : \mu\alpha.[] \times \mu\alpha.[\texttt{value} : (\texttt{String}, 1/1), \texttt{next} : (\alpha, 0/0)], 0 \to \texttt{Int}, 0$$

contains the types $t_{\texttt{list}}$ and `Int` as types for the parameter and the result. The additional implicit parameter `this` is typed as empty object with $\mu\alpha.[]$, since `upload_list` is not called as method. The annotation 0 on the parameter side state that the function does not require any constant number of reserved resource units to be called and the annotation on the result side states that no reserved resource units are returned by the function call.

The reserved resource units provided to `upload_list` via its parameter `list` are consumed by the function body. Therefore, a function call to `upload_list` has to update the amortised types of the parameters at the call site.

Example 6.2.2.

```
1 var x = {value: "1.jpg", next: null};
2 upload_list(x)
```

Assume the variable x has a type

$$t_x = \mu\alpha.[\texttt{value} : (\texttt{String}, 2/2), \texttt{next} : (\alpha, 0/0)]$$

before line 2. After line 2 it is reduced to

$$t'_x = \mu\alpha.[\texttt{value} : (\texttt{String}, 1/2), \texttt{next} : (\alpha, 0/0)],$$

so that the reserved resource units consumed by the function cannot be reused. Due to the remaining annotation $(1/2)$, the remaining list has enough reserved resource units to call `upload_list` or a similar function once more.

AmorJiSe manages this reduction of the annotations by a sharing relation

$$t \hookrightarrow t' \oplus t''.$$

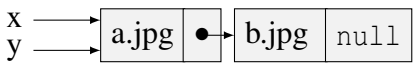
This constraint splits the type t into two structurally equivalent types, where the reserved resource units previously contained in t are now distributed between the two types t' and t''. Therefore, an invariant of this relation is that the potential $\Sigma v : t$ of an arbitrary typed value v is equal to the sum of the potentials $\Sigma v : t' + \Sigma v : t''$. After the split, the type t' is used to perform the computation, in this case the function call, while the type t'' is stored back into the context Γ for future use of the variable x.

Aliasing poses additional challenges:

Example 6.2.3.

```
1 var x = {value: "a.jpg", next: null};
2 var y = x;
3 x.next ={value: "b.jpg", next: null};
4 upload_list(y)
```

The variables x and y in this example are both objects with the data type $\mu\alpha.[\texttt{value} : \texttt{String}, \texttt{next} : \alpha]$. Line 2 aliases the variables x and y resulting in the following memory layout:



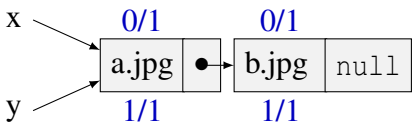
Line 1 and 3 of this example populate the list through `x`, but line 4 accesses the list via the alias `y`. The amortised type system has to make sure that the resource units reserved by the amortised annotations are

1. stored in the data structure while writing to `x`,
2. accessible through `y`,
3. after the consumption not accessible through `x` anymore.

For this reason, the second annotation N, called the *capacity*, tracks the sum of resource units associated with all aliases of the stored value. When writing to a value, enough resource units need to be provided to satisfy all units associated with the aliases. For example, the type for the field `value` of `x` in line 1 is $\texttt{value}{:}(\texttt{String}, 1/1)$. According to the annotation $/1$, each assignment to `value`, e.g. in line 1, requires 1 resource unit. This accounts for the resource unit reserved with the element "a.jpg" according to the annotation $1/$. In the assignment in line 2 the associated resource unit is transferred to `y`, once again using the sharing relation, to make it available for the resource consumption in line 4. This results in the following types:

$$\texttt{x} \mapsto \mu\alpha[\texttt{value} : (\texttt{String}, 0/1), \texttt{next} : (\alpha, 0/0)]$$
$$\texttt{y} \mapsto \mu\alpha[\texttt{value} : (\texttt{String}, 1/1), \texttt{next} : (\alpha, 0/0)]$$

The annotation $1/1$ in the type for `y` ensure that the resource units can be used via `y` but not through `x` with the annotation $0/1$. However, assigning to `x` still requires 1 resource unit to be provided. Accordingly, line 3 consumes 1 resource unit to cover the resource unit associated with the alias `y`. This results in the following memory layout and associated resource units:



These annotations allow `y` to be used as the parameter for `upload_list` in line 4. The call consumes the reserved resource units associated with `y` and reduces the type to

$$y \mapsto \mu\alpha[\texttt{value} : (\texttt{String}, 0/1), \texttt{next} : (\alpha, 0/0)]$$

The analysis has to ensure that for a set of aliases $\{\mathtt{x},\mathtt{y},\ldots\}$ with the annotations $(n_\mathtt{x}/N_\mathtt{x}),(n_y/N_y),\ldots$ each capacity is bigger than the sum of resource units stored with all potential aliases:

$$\left(\sum_{i=\mathtt{x},\mathtt{y},\ldots} n_i\right) \leq N_\mathtt{x}, \left(\sum_{i=\mathtt{x},\mathtt{y},\ldots} n_i\right) \leq N_\mathtt{y}, \ldots$$

One extreme case of aliases are loops in a data structure. They present a difficult problem for amortised systems. Take, for example, a recursive list represented as above as $\mu\alpha.[\mathtt{value} : (\mathtt{String}, 1/1), \mathtt{next} : (\alpha, 0/0)]$. The type of the field `value` has 1 reserved resource unit associated with it, which means the total number of units contained in the list is equal to its length. The expression `list.next=list` assigns the head of the list as the successor of the head and creates a loop. If the loop contains a positive number of reserved resource units, the loop now stores an infinite amount of resources in total. The typing rules prohibit this case automatically by indirectly requiring that the remaining amortised annotations nested within the potential loop are all 0. This way, code producing a loop can be typed, but they cannot carry reserved resource units in the recursive part of the type.

The example considered so far used objects equivalent to classes in other programming languages with a static set of fields. JavaScript, however, can manipulate the set of fields dynamically during runtime. Assignment to an unassigned field extends an object to contain this new field. Together with aliasing, this poses an additional challenge:

Example 6.2.4.

```
1 var x = {next:null};
2 var y = x;
3 y.value = "a.jpg";
4 x.next = {value:"b.jpg",next:null};
5 upload_list(y)
```

In comparison to Example 6.2.3, here the `value` field is added to the list object via `y`. Since `upload_list` is called with the parameter `y` in line 5, `y` must by typed with at least the annotation $1/1$. This information is needed during the assignment to `x` in line 4 to account for the reserved resource unit stored with the path `y.next.value`. However, the simple type

$$t_0 = \mu\alpha.[\mathtt{next} : (\alpha, 0/0)]$$

does not capture any information about `value` and the type

$$t_0 = \mu\alpha.[\texttt{value} : (\texttt{String}, 1/1), \texttt{next} : (\alpha, 0/0)]$$

wrongly claims that the field `value` is already assigned. Following the approach in JS_0^T [9], AmorJiSe marks existing fields in object types with the marker $\bullet$ as definite and adds potential fields with the marker $\circ$. Definite ($\bullet$) fields are guaranteed part of the value and can therefore be written to and read. Potential ($\circ$) fields, in contrast, might be added to the object later. Consequently, they can be written, but not read. Inserting these markers yields the type

$$t = \mu\alpha.[\texttt{value} : ((\texttt{String}, 1/1), \circ), \texttt{next} : ((\alpha, 0/0), \bullet)]$$

for `x` in line 1. It states that the value stored at `x` can be extended by a field `value` with the annotation $1/1$. Like the annotation in Example 6.2.3 this potential annotation $1/1$ is split in line 2 to reflect that the potentially associated resource unit can only be used via the alias `y`. This way, AmorJiSe manages controlled object extensions.

Finally, the evaluation relation has to be resource-aware to proof the bounds resulting from the amortised annotations. in AmorJiSe it looks as follows:

$$e, H, \chi \xrightarrow[n']{n} e', H', \chi'.$$

It describes that in the heap H and scope χ the expression e can be evaluated with n provided resource units and after execution n' resource units are still available.

In summary, AmorJiSe takes existing data types and inserts amortised annotations into the following 4 different locations.

1. AmorJiSe extends each type with numeric annotations:

   $$t^+ := (t, n/N).$$

   The first annotation n describes how many resource units are reserved with a value typed with t^+ and N describes how many units have to be reserved when writing to a value typed as t^+ to cover the resources reserved with aliases.

2. It annotates the evaluation relation with 2 numbers:

   $$e, H, \chi \xrightarrow[n']{n} e', H', \chi'.$$

   The first annotation n describes how many resource units the execution of the expression e to e' requires. The second annotation n' asserts that n' of those units will be returned and can be reused after this execution. However, during the evaluation of e, these returned units might have been used temporarily.

3. It annotates the typing relation similar to the evaluation relation:

$$\Gamma, n \vdash e : t | \Gamma', n'.$$

This certifies that in the context Γ with n resource units the expression e evaluates to a value of type t with the resulting context Γ' and n' units which can be reused after the execution. Additional annotations are embedded into the types in the contexts Γ and Γ' as described above.

4. It annotates each function type:

$$O \times t, n \to t', n'.$$

The annotations assert that each call to the function requires n resource units and n' resource units become free after the function execution has finished. Additionally, the parameter types O, t and t' include annotations as described above.

AmorJiSe infers minimal values for all annotations contained in the types and the typing judgement. To achieve this, it assumes that data types for the analysed code are provided. Into those data types AmorJiSe inserts annotation variables to promote them to amortised types and then infers a set of linear constraints of the form $\sum_{i=1..k} n_i \leq \sum_{i=1..k'} n'_i$ over the annotation variables. The solution for this constraint system results in a valid amortised typing and final resource bounds for the execution of the code can be extracted from them.

6.3 The AmorJiSe system

The following section formally presents the type system AmorJiSe with all needed notations and properties. The main result is the type soundness: the resource bounds derived from the resulting types limit the resource usage of the formal evaluation of the typed expression. An inference algorithm to automatically obtain the amortised annotations for a given typing is also presented below.

6.3.1 Basic Definitions

Let there be a set *Addresses* of valid addresses ι, a set *VarNames* of valid variable names *var* and a set *FieldNames* of valid names m for object fields. Furthermore, let there be a set *Types* of all possible types which, in the following discussion, will be

instantiated differently for the underlying type system and for AmorJiSe. To prevent ambiguity, assume $\mathbb{N} = \{0,1,2,...\}$.

Definition 6.3.1. The set *Values* contains the following kinds of valid values:

$$\begin{aligned} Bool &= \{\texttt{true}, \texttt{false}\} \\ Int &= \{-2^{1023}, ..., 2^{1023}\} \cup \{\texttt{NaN}\} \\ Fun &= \{\texttt{function} f(var)\{e\} \mid f \in VarNames, e \text{ is an expression}\} \\ Obj &= \{\ll m_1 : v_1, ..., m_k : v_k \gg \mid k \in \mathbb{N}, m_{1..k} \in FieldNames, v_{1..k} \in Values\} \\ Values &= Bool \cup Int \cup Addresses \cup Fun \end{aligned}$$

Definition 6.3.2. The object value $o = \ll m_1 : v_1, ..., m_k : v_k \gg \in Obj$ represents a map, mapping each field name $m \in FieldNames$ to a value $v \in Values$:

$$o(m) = \begin{cases} v_i & \text{if } m = m_i \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

Definition 6.3.3.

1. A *heap* H is a map $Addresses \to Obj \cup \{\mathcal{U}df\}$ mapping each heap address to a value or the special value $\mathcal{U}df$ which indicates that an address has not been assigned yet.
2. A *scope* χ is a map $VarNames \to Values \cup \{\mathcal{U}df\}$ mapping each variable name to its current value.
3. A pair (χ, H) is called a *runtime state*
4. A *context* Γ is a map $VarNames \to Types \cup \{\texttt{None}\}$ mapping each variable name to its current type or `None` if its type is unknown.

Definition 6.3.4.

For a map M and a value v the standard operators are defined as follows:

1. $M(m)$ returns the value mapped from m by M
2. $M[m \mapsto v]$ defines a map M' resembling M except for the the value $M'(m)$:

$$M(m')' = \begin{cases} v & \text{if } m' = m \\ M(m') & \text{otherwise} \end{cases}$$

3. $v[\alpha/v']$ replaces all unbound occurrences of α in v by v'.

6.3.2 Data types

AmorJiSe builds upon existing data types for JavaScript. Candidates for the underlying system have been discussed in the previous chapter. AmorJiSe assumes that the underlying system describes the data structures of the analysed expressions. In particular, for an expression e consider all evaluations $H_{e'}, \chi_{e'}, e' \twoheadrightarrow H'_{e'}, \chi'_{e'}, v_{e'}$ of subexpressions e' which occur in the derivation of the evaluation $H, \chi, e \twoheadrightarrow H', \chi', v$. AmorJiSe assumes the underlying system provides

- a type $t_{e'}$ for each such e' (including e itself)
- contexts $\Gamma_{e'}, \Gamma'_{e'}$ providing types for the variables occurring in each runtime state $(\chi_{e'}, H_{e'})$ and $(\chi'_{e'}, H'_{e'})$

such that $\Gamma_{e'} \vdash e' : t_{e'} : \Gamma'_{e'}$.

Definition 6.3.5. Let there be a set *ObjectTypes* of object types O with the following properties.

1. An object type O maps each $m \in \mathit{FieldNames}$ to a pair (t, ft) of a type t and a *field state* $\mathit{ft} \in \{\circ, \bullet\}$. An object type O mapping the field names $m_1, ..., m_k$ (and all other m to $(\mathcal{U}df, \circ)$) is in the following represented as

$$\mu\alpha.[m_1 : (t_1, \psi_1), ..., m_k : (t_k, \psi_k)]$$

$$\text{with } k \in \mathbb{N}, m_{1..k} \in \mathit{FieldNames}, t_{1..k} \in \mathit{Types} \cup \{\alpha\}$$

2. For each object type O
   (a) the operator $\texttt{Dom}(O)$ returns all fields m mapped by O to a pair other than $(\mathcal{U}df, \circ)$
   (b) the read operator $O(m)$ returns the pair $(t_i[\alpha/O], \psi_i)$ with $m = m_i$

The $\mu\alpha$ operator in front of the object type enables the specification of recursive object types, which introduces infinite types into the data type system. Each field in an object type O is marked via its field state ψ as either $\bullet$ or $\circ$. The field state has the following intuition. A field which is marked as $\bullet$ has been assigned and can therefore be read or written. If a field $O(m)$ is marked as $\circ$, the object O does not contain the field m but can be extended by this field in the future. Therefore, the field m can only be written. If the underlying system does not include the markers $\circ/\bullet$, this information

can usually be inferred from the existing fields by marking all present fields in an object type as • and all remaining accessed fields as ∘.

The object types of AmorJiSe do not use any δ field (see Section 5.2.1) in the row types themselves, but the object field lookup $O(m)$ could easily be extended to internally map unknown fields to a δ-field.

Definition 6.3.6. Let there be a set *FunctionTypes* of function types G, represented as $O \times t_{\mathrm{x}} \to t_{ret}$ with

1. the object type O for the receiver `this`
2. the type t_{x} for the parameter `x`
3. the type t_{ret} for the value returned by the function body

Definition 6.3.7. For the underlying type system, the set *Types* - in the following called data types - is defined as follows

$$Types = \{\texttt{Int}, \texttt{Bool}\} \cup ObjectTypes \cup FunctionTypes$$

As primitive types, the rules presented here only consider `Int` and `Bool`, but adding further terminal types does not pose an additional challenge.

The precision of the data types influences the results of AmorJiSe. Consider for example heap space as a resource in the following example

```
1 o={value:42};
2 o.value=1;
```

Line 2 only overwrites the `value` field of the object `o`. Therefore, line 2 does not allocate any heap space. Its actual resource consumption is 0. The object `o` prior to the overwrite operation can be typed with either of the data types

$$t_{\circ} = [\texttt{value} : (\texttt{Int}, \circ)]$$
$$t_{\bullet} = [\texttt{value} : (\texttt{Int}, \bullet)].$$

In any case the writing operation (line 2) makes the field `value` definite in the resulting type for `o`:

$$t' = [\texttt{value} : (\texttt{Int}, \bullet)]$$

. However, with the type $t_{\circ}$ no guarantee is given that the field is present in the object `o` before line 2. Assigning to `value` might be an object extension which consumes an

additional heap cell. Therefore, AmorJiSe has to assume a resource consumption of 1 unit, which is a correct but imprecise bound for the resource consumption. In contrast, the type $t_\bullet$ guarantees that the field `value` is already present in `o` and AmorJiSe can infer that no additional resources are consumed. The more precise data type $t_\bullet$ leads to a more precise description of the resource consumption of this example.

6.3.3 JavaScript Syntax

Figure 6.1 shows the subset of JavaScript covered by AmorJiSe. It constitutes a core language focused on the object behaviour of JavaScript which differentiates JavaScript from other languages. For simplicity, this core language only contains integer and Boolean constants, but additional primitive constants can be defined analogously. Concerning functions, AmorJiSe only considers non-nested function statements. Function expressions and nested function statements can be handled similarly. All functions can be called as constructor, function or method as usual in JavaScript. As variables, AmorJiSe only considers `this` and the parameter `x`. However, the type context and scope defined later are powerful enough to also represent the JavaScript variable definition using the *var*keyword. As control structures, only the conditional expression $e_1?e_t : e_f$ is considered. The `if`-statement can be handled equivalently. Loops are discussed as an extension later in Section 6.7 and 6.8.

6.3.4 Resource model

The resource model in AmorJiSe describes how the analysed resource is accessed. The model consists of two parts, outlining the resource costs for API activated and language activated resources (see Section 2.2) separately. The API activated resources are modelled by defining the resource consumption of the APIs in the initial context. They are specified using the standard function types, which will be presented in more detail below. Using this part of the model, each resource model defined for PhoneWrap can be directly translated into a AmorJiSe model. For example, the resource model describing the message service given by

```
1 guard : ["smsplugin.send"]
```

can be described by setting

$$\Gamma_0(\texttt{smsplugin.send}) = (((O \times t_x, 1 \to t_{ret}, 0), 0/0), \bullet)$$

Figure 6.1 Syntax

$P ::= F^*; e$	(Program)
$F ::= \texttt{functionf(x)}\{e\}$	(FuncDecl)
$e ::=$	(Expression)
$\quad var$	(variables)
$\quad \texttt{f}$	(function identifier)
$\quad \texttt{new f}(e)$	(constructor call)
$\quad e; e$	(sequence)
$\quad e.m(e)$	(member call)
$\quad e.m$	(member read)
$\quad \texttt{f}(e)$	(function call)
$\quad lhs = e$	(assignment)
$\quad e?e:e$	(conditional)
$\quad \texttt{null}$	(null)
$\quad \texttt{n,true,false}$	(value literal)
$var ::= \texttt{this,x}$	(variables)
$lhs ::= \texttt{x} \mid e.m$	(LeftHandSide)

in the initial context Γ_0 with the appropriate types $O, t_{\texttt{x}}, t_{ret}$ for the parameters and return value. It describes the resource consumption behaviour with the annotations 1 and 0 as well as the fact that it is available to be called ($\bullet$) and does not carry any resources ($0/0$).

Language activated resources in the model are specified via the consumption constants:

Definition 6.3.8.

- For each language construct r let there be a consumption constants c_r describing the resource consumption of one execution of r. For example, the constant $c_{(\text{VARW})}$ describes the resource usage of a write operation to a variable.

- For the field write operation let there be two consumption resources $c_{(\text{MEMW})}(true)$ and $c_{(\text{MEMW})}(false)$ describing the resource consumption of a write operation to a pre-existing field $(true)$ or of a write operation to a new field $(false)$ of an object. Assume $c_{(\text{MEMW})}(false) \geq c_{(\text{MEMW})}(true)$ for these constants.

Every syntax construct in the language has been given a separate consumption constant. This is the most precise resource model one can achieve on the syntactic level.

Constants for constructs which are not important for a particular resource model can be set to 0.

The constant $c_{(\text{MEMW})}$ describing the resource consumption of a field write to an object has a special role: in JavaScript a write operation to an object field could either be an overwrite operation or an object extension. These two operations cannot be distinguished syntactically, but might consume different resources, e.g., heap space. For this reason, there exists one constant $c_{(\text{MEMW})}(true)$ which is used when the assigned field was available before the assignment and a second constant $c_{(\text{MEMW})}(false)$ which is used otherwise. For the soundness proof later, the assumption $c_{(\text{MEMW})}(false) \geq c_{(\text{MEMW})}(true)$ is required. The assumption is reasonable, since over-writing an existing field usually is at most as expensive as creating a new field by writing to a non-existing field.

These constants are used in the typing rules for AmorJiSe as well as in the annotated operational semantics.

Example 6.3.9. For the usual resource models most of the constants c_r are set to 0. For example, to define a resource model where each newly created field of an object consumes one resource unit, the constants are set as $c_{memW}(false) = 1$ and $c_r = 0$ for all other constants r.

6.3.5 Operational semantics

An expression e, composed from the provided syntax, can be evaluated as stated by the evaluation relation

$$e, H, \chi \xrightarrow[n']{n} v, H', \chi'$$

which reduces e to a value v from the set $Values \cup \{\mathcal{U}df\}$. The evaluation relation states that in the heap H and scope χ the expression e is evaluated to the value v resulting in the new heap H' and the new scope χ'. Furthermore, it specifies that the evaluation can be performed with n available resource units, of which n' are returned after the execution to be reused. The annotation n' can be used to specify the behaviour of reusable resource kinds like open file handles or memory space.

6.3.5.1 Evaluation rules

The semantics of the core language is defined via the big step evaluation rules in Figure 6.2. In addition, the resource model assumes the evaluation relation $f, H, \chi \xrightarrow[n']{n} v, H', \chi'$

provides appropriate function values v for all function names f defined in the resource model as $\Gamma(f) = (((O \times t_x, n \to t_{ret}, n'), n_f/N_f), \bullet)$.

Note that in these rules the consumption constant is always added to the resource requirements of the rule. Therefore the resource units are always required before the whole expression is evaluated. In some cases, tighter bounds could be computed by allowing the resource units required to be generated by the evaluation of a sub-expression. For example, in the rule (S-VARW) the assignment is executed after the right-hand-side expression e is evaluated. Therefore, the evaluation of e could generate the resource requirements by its annotation n'_e. An alternative version of this rule, respecting this, is:

$$\frac{e, H, \chi \xrightarrow[n'_e + c_{(\text{VARW})}]{n_e} \twoheadrightarrow v, H', \chi_1 \qquad \chi' = \chi_1[var \mapsto v]}{var = e, H, \chi \xrightarrow[n'_e]{n_e} \twoheadrightarrow v, H', \chi'} \qquad \text{(S-VARW-ALT)}$$

All evaluations computed with the original rule `S-val` are also valid computations with this alternative rule. However, the alternative rule could in some cases evaluate an expression $var = e$ with less required resource units, if $n'_e \geq n_e$. The consistent placement of the constants in the requirement of the full expression, as chosen in the rules in Figure 6.2, simplifies the soundness proof while generating correct bounds for real world JavaScript behaviour.

The rules use the resource addition operator $(n, n') = (n_1, n'_1) \fatsemi (n_2, n'_2)$ to compute the resource behaviour of the sequential execution of two expressions $e_1; e_2$. It is defined in the following way:

Definition 6.3.10. Given annotation pairs (n_1, n'_1) and (n_2, n'_2) define the resource addition operators $(n, n') = (n_1, n'_1) \fatsemi (n_2, n'_2)$ as follows:

$$n = \begin{cases} n_1 & \text{if } n'_1 \geq n_2 \\ n_1 - n'_1 + n_2 & \text{if } n'_1 < n_2 \end{cases}$$
$$n' = \quad n - (n_1 - n'_1 + n_2 - n'_2)$$

Figure 6.3 illustrates examples for the two cases of this definition. In the first case e_1 returns more resource units n'_1 then e_2 requires n_2. Therefore, e_2 is executed with the resources returned by e_1 and the concatenation $e_1; e_2$ can be executed with n_1 provided resource units. In the second case ($n'_1 < n_2$) the expression e_2 requires more resource

Figure 6.2 Big-step operational semantics

$$\frac{c \in Int \cup Bool}{c, H, \chi \xrightarrow[0]{c_{(\text{VAL})}} \!\!\!\!\rightarrow c, H, \chi} \quad \text{(S-VAL)}$$

$$\frac{}{var, H, \chi \xrightarrow[0]{c_{(\text{VARR})}} \!\!\!\!\rightarrow \chi(var), H, \chi} \quad \text{(S-VARR)}$$

$$\frac{\begin{array}{c} e, H, \chi \xrightarrow[n'_e]{n_e} \!\!\!\!\rightarrow v, H', \chi_1 \\ \chi' = \chi_1[var \mapsto v] \end{array}}{var = e, H, \chi \xrightarrow[n'_e]{n_e + c_{(\text{VARW})}} \!\!\!\!\rightarrow v, H', \chi'} \quad \text{(S-VARW)}$$

$$\frac{e, H, \chi \xrightarrow[n'_e]{n_e} \!\!\!\!\rightarrow \iota, H', \chi'}{e.m, H, \chi \xrightarrow[n'_e]{n_e + c_{(\text{MEMR})}} \!\!\!\!\rightarrow H'(\iota)(m), H', \chi'} \quad \text{(S-MEMR)}$$

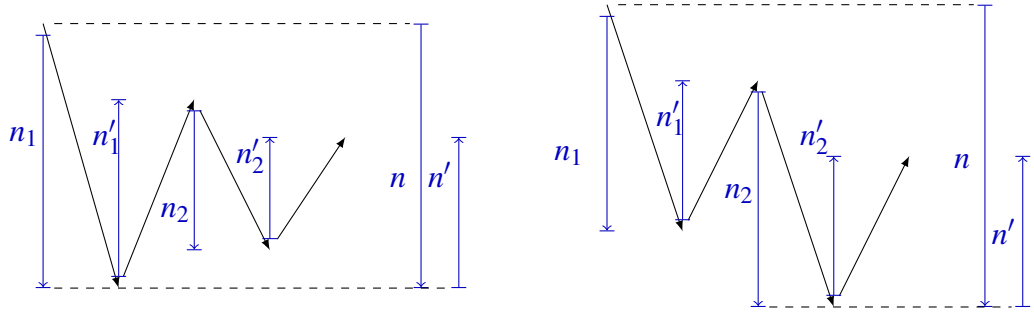
$$\frac{\begin{array}{c} e_1, H, \chi \xrightarrow[n'_1]{n_1} \!\!\!\!\rightarrow \iota, H_1, \chi_1 \\ e_2, H_1, \chi_1 \xrightarrow[n'_2]{n_2} \!\!\!\!\rightarrow v, H_2, \chi' \\ H' = H_2[\iota \mapsto H_2(\iota)[m \mapsto v]] \\ (n_\S, n'_\S) = (n_1, n'_1) \S (n_2, n'_2) \end{array}}{e_1.m = e_2, H, \chi \xrightarrow[n'_\S]{n_\S + c_{(\text{MEMW})}(m \in \text{Dom}(H_2(\iota)))} \!\!\!\!\rightarrow v, H', \chi'} \quad \text{(S-MEMW)}$$

$$\frac{\begin{array}{c} e_b, H, \chi \xrightarrow[n'_1]{n_1} \!\!\!\!\rightarrow true, H_1, \chi_1 \\ e_t, H_1, \chi_1 \xrightarrow[n'_2]{n_2} \!\!\!\!\rightarrow v, H', \chi' \\ (n_\S, n'_\S) = (n_1, n'_1) \S (n_2, n'_2) \end{array}}{e_b ? e_t : e_f, H, \chi \xrightarrow[n'_\S]{n_\S + c_{(\text{COND})}} \!\!\!\!\rightarrow v, H', \chi'} \quad \text{(S-CONDTRUE)}$$

$$\frac{\begin{array}{c} e_1, H, \chi \xrightarrow[n'_1]{n_1} \!\!\!\!\rightarrow false, H_1, \chi_1 \\ e_3, H_1, \chi_1 \xrightarrow[n'_2]{n_2} \!\!\!\!\rightarrow v, H', \chi' \\ (n_\S, n'_\S) = (n_1, n'_1) \S (n_2, n'_2) \end{array}}{e_1 ? e_2 : e_3, H, \chi \xrightarrow[n'_\S]{n_\S + c_{(\text{COND})}} \!\!\!\!\rightarrow v, H', \chi'} \quad \text{(S-CONDFALSE)}$$

$$\frac{\begin{array}{c} e_1, H, \chi \xrightarrow[n'_1]{n_1} \!\!\!\!\rightarrow v_1, H_1, \chi_1 \\ e_2, H_1, \chi_1 \xrightarrow[n'_2]{n_2} \!\!\!\!\rightarrow v_2, H', \chi' \\ (n_\S, n'_\S) = (n_1, n'_1) \S (n_2, n'_2) \end{array}}{e_1; e_2, H, \chi \xrightarrow[n'_\S]{n_\S + c_{(\text{SEQ})}} \!\!\!\!\rightarrow v_2, H', \chi'} \quad \text{(S-SEQ)}$$

Figure 6.3 $(n,n') = (n_1,n_1') \dot{\S} (n_2,n_2')$



(a) $n_1' > n_2$

(b) $n_1' < n_2$

units than e_1 returns. Therefore $n_2 - n_1'$ resource units are required in addition to the n_1 units required by the first expression.

All annotations n,n' in the evaluation rules are implicitly assumed to be non-negative. The addition operator $n,n' = (n_1,n_1') \dot{\S} (n_2,n_2')$ respects this invariant:

Lemma 6.3.11. Assume $n_1,n_1',n_2,n_2' \geq 0$ and $(n,n') = (n_1,n_1') \dot{\S} (n_2,n_2')$. Then it holds $n,n' \geq 0$

Proof. According to the definition of the resource addition there are two different cases:

Case $n_1' \geq n_2$: In this case it holds $n = n_1 \geq 0$ directly. For n' consider:

$$\begin{aligned} n' &= n - n_1 + n_1' - n_2 + n_2' \\ &= n_1 - n_1 + n_1' - n_2 + n_2' \\ &= n_1' - n_2 + n_2' \\ &\geq n_2 - n_2 + n_2' \\ &= n_2' \geq 0 \end{aligned}$$

Case $n_1' \leq n_2$: In this case n and n' can be computed as

$$\begin{aligned} n &= n_1 - n_1' + n_2 \\ &\geq n_1 - n_2 + n_2 \\ &= n_1 \geq 0 \\ n' &= (n_1 - n_1' + n_2) - n_1 + n_1' - n_2 + n_2' \\ &= n_2' \geq 0 \end{aligned}$$

□

Figure 6.4 Types syntax

$$
\begin{array}{rll}
t ::= & O \mid G \mid \texttt{Int} \mid \texttt{Bool} \mid \alpha \mid \texttt{None} & \text{(pre-type)} \\
t^+ ::= & (t, n/N) & \text{(full type)} \\
O ::= & \mu\alpha.M \mid M & \text{(object)} \\
M ::= & [(m : tm)^*] & \text{(memberlist)} \\
tm ::= & (t^+, \psi) & \text{(membertype)} \\
\psi ::= & \bullet \mid \circ & \text{(Field state)} \\
G ::= & O^+ \times t^+, n \to t^+, n & \text{(functions)}
\end{array}
$$

where $n, N \in \mathbb{N}^{\geq 0}$ and m ranges over strings

Lemma 6.3.12. Given an evaluation $e, H, \chi \xrightarrow[n']{n} v, H', \chi'$ the annotations n, n' are unique.

Proof. The proof is an induction on the length of the derivation of $e, H, \chi \xrightarrow[n']{n} v, H', \chi'$. The derivation is of length 1 if e is constant (S-VAL) or a variable (S-VARR). In both cases, the annotations are uniquely determined as $(c_{(\text{VAL})}, 0)$ or $(c_{(\text{VARR})}, 0)$. In all other cases, the induction hypothesis applies to all subexpressions. Either the final annotation for e is directly determined by the annotation of one of its subexpressions or is calculated from multiple subexpression annotations via the $;$ operator which is deterministic. This proves the claim. □

6.3.6 Types

6.3.6.1 Definition

Figure 6.4 defines the syntax of AmorJiSe's types and, therefore, the set *Types* for AmorJiSe. A full type t^+ consists of a pre-type t and an amortised annotation n/N. A pre-type t is either a basic type (here only `Int` or `Bool`, but extendable, for example, to `String`), an annotated object O or an annotated function G. Furthermore, AmorJiSe considers the types `None` to indicate a type which should not be accessed and α to represent recursion in object types.

Object types $O = \mu\alpha.M$ consist of a recursive binder $\mu\alpha$ and a row type M. Each free occurrence of α within the row type M is a placeholder for the whole type O. The

row type M is represented as a list of fields with their full types and markers $\circ/\bullet$. Field lookup always unfolds a recursive object type before returning the field-type:

Definition 6.3.13. Given an object type O and a field m define the lookup operator $O(m)$ as

$$O(m) = \begin{cases} (t^+[\alpha/O], ft) & \text{if } O = \mu\alpha.[\ldots, m : (t^+, ft), \ldots] \\ ((\texttt{None}, 0/0), \circ) & \text{otherwise} \end{cases}$$

Function types G consist of a full object type $O^+ = (O, n/N)$ describing the type of the expected receiver value `this`, and full types t^+ for the parameter `x` and the return value. Furthermore, the function type describes the resource requirement n for the function and the resource units returned by the function n'.

Intuitively, the only difference between the amortised types of AmorJiSe and the data types of the underlying system are the amortised annotations n/N. They consist of two parts with the following intuitive meaning: Reading from a value with annotation n/N can retrieve up to n reserved resource units and writing to it consumes N units to account for the units which can be read from this value later through all available aliases.

To speak about the different parts of a type $(t, n/N)$ the *projections* extract the pre-type t, the units n and the capacity N.

Definition 6.3.14.

$$[\![(t, n_t/N_t)]\!]^t = t$$
$$[\![(t, n_t/N_t)]\!]^n = n_t$$
$$[\![(t, n_t/N_t)]\!]^N = N_t$$

In general, to avoid notational clutter, identify marked types (t, ψ) with the contained full type t within all the defined relations and notations. In particular, for the projection identify

$$[\![(t^+, \psi)]\!]^t = [\![t^+]\!]^t$$
$$[\![(t^+, \psi)]\!]^n = [\![t^+]\!]^n$$
$$[\![(t^+, \psi)]\!]^N = [\![t^+]\!]^N$$

and for typed values identify

$$v : (t^+, \psi) = v : t^+$$

During type inference, AmorJiSe converts the data types into amortised types by inserting annotation variables into all necessary places. The following notations use the operator $\mathcal{T}(e)$ to obtain the data type of the expression e with injected annotation variables.

Definition 6.3.15. Given an expression e and its data type t let $\mathcal{T}(e)$ be defined as $\mathcal{T}(e) = [\![t]\!]^+$ with

$$\begin{aligned}
[\![\texttt{Int}]\!]^+ &= (\texttt{Int}, n_1/n_2) \\
[\![\texttt{Bool}]\!]^+ &= (\texttt{Bool}, n_1/n_2) \\
[\![\alpha]\!]^+ &= (\alpha, n_1/n_2) \\
[\![\mu\alpha.[m_1 : (t_1, \psi_1), \ldots, m_k : (t_k, \psi_k)]]\!]^+ &= \mu\alpha.[m_1 : ([\![t_1]\!]^+, \psi_1), \ldots, m_k : ([\![t_k]\!]^+, \psi_k)] \\
[\![O \times t_x \to t']\!]^+ &= [\![O]\!]^+ \times [\![t_x]\!]^+, n_1 \to [\![t']\!]^+, n_2
\end{aligned}$$

In each case the inserted annotation variables n_i are assumed to be fresh variables.

In some rules a type-lookup via $\mathcal{T}(\cdot)$ has to be specified by more than the expression e. For example, while reading the type for a variable `x` from the context Γ, three different types are available for the variable `x`: the type of `x` in the context before the variable is read, the type of `x` in Γ after the variable is read and the type of the resulting value $\Gamma(\texttt{x})$ used for the subsequent computation. To differentiate between those three types, the typing rules use the notation $\mathcal{T}(e,1), \mathcal{T}(e,2), \mathcal{T}(e,3)$, which each produce an annotated type $[\![t]\!]^+$ for a valid type t for the expression e with distinct annotation variables. The specific semantics of each annotated type will be clear from the typing rule. As in the example, the three types $\mathcal{T}(e,1), \mathcal{T}(e,2), \mathcal{T}(e,3)$ might be based on the same data type in the underlying system. Reading `x` might result in the types $\mathcal{T}(\texttt{x},1) = (\texttt{Int}, n_1/N_1)$, $\mathcal{T}(\texttt{x},2) = (\texttt{Int}, n_2/N_2)$ and $\mathcal{T}(\texttt{x},3) = (\texttt{Int}, n_3/N_3)$ all based on the underlying type $t_{\texttt{x}} = \texttt{Int}$. However, the annotations are different. In this case, the data type $\mathcal{T}(e)$, here `Int`, is copied three times and different annotations inserted into each copy.

6.3.6.2 Typing context

Definition 6.3.16. Extending Definition 6.3.3 a type context Γ in AmorJiSe is a map $\mathit{VarNames} \to (\mathit{Types} \cup \{\texttt{None}\}) \times \{\circ, \bullet\}$, which maps variable names *var* to a full type t^+ and a marker ψ.

Variables in JavaScript can be declared with the `var` statement. JavaScript parses each scope block of the code in two passes. The first pass executes only `var` statements and injects the declared variables into a fresh scope frame. The second pass executes the whole block in this initialised scope frame, which means all declared variables can be assigned. If a declared variable is read before a value is assigned, JavaScript returns the value `undefined` which might potentially lead to unexpected behaviour.

The AmorJiSe context Γ mimics this behaviour and allows to store uninitialised variables marked as potential with $\circ$. Once a variable marked with $\circ$ has been assigned, AmorJiSe changes the mark to $\bullet$ and records that this variable can be read. This has the additional advantage that contexts can be treated equivalent to object types just as JavaScript scope frames are treated as an ordinary object by the ECMA standard [61, Section 9.2]. This way, it is very easy to extend AmorJiSe with JavaScript's `var` keyword and the `with` statement, which inserts an arbitrary object into the scope chain.

6.3.6.3 Type judgement

The main judgement of AmorJiSe is

$$\Gamma, n \vdash e : \mathcal{T}(e) | \Gamma', n'$$

with $n, n' \in \mathbb{N}^{\geq 0}$. It expresses that in the typing context Γ given n resource units the expression e is given the full type $\mathcal{T}(e)$, the context after the evaluation of e is Γ' and n' resource units are available after the evaluation.

Intuitively, the validity of $\Gamma, n \vdash e : \mathcal{T}(e) | \Gamma', n'$ directly implies all judgements of the form $\Gamma, n+k \vdash e : \mathcal{T}(e) | \Gamma', n'+k$ for $k \in \mathbb{N}$. If an expression can be evaluated with n resource units, additional resource units k are simply ignored by the expression. This claim can be formally proven via induction on the typing rules provided in Figure 6.5. This proof is omitted here since it does not add to the desired properties of the system.

Note that the opposite with $k \in -\mathbb{N}$ might not be true, even if $n, n' > 0$, as the expression might need some resource units temporarily and free them afterwards. Consider, for example, the resource of open file handles. An expression could open one file, write data into it and then close the file properly. This expression would need one resource unit to be executed and would return this resource unit after the execution. Therefore, $\Gamma, 1 \vdash e : \mathcal{T}(e) | \Gamma', 1$ holds true for this expression e, but not $\Gamma, 0 \vdash e : \mathcal{T}(e) | \Gamma', 0$.

6.3.6.4 Full types versus pre-types

The type $\mathcal{T}(e)$ in the judgement above is a full type of the form $(t, n_e/N_e)$ instead of the simple pre-type t. In the typing judgement with this full type $\Gamma, n \vdash e : (t, n_e/N_e))|\Gamma', n'$ the annotation n_e is redundant since $\Gamma, n \vdash e : (t, 0/N))|\Gamma', n' + n_e$ expresses the same number of available resource units. However, the annotation N_e is necessary: if e evaluates to a reference into the heap, every following assignment to this reference needs to provide enough resource units to satisfy the resource units retrievable through the aliases of this heap location.

For example, in the expression e :`y.value="a.jpg"` (see Example 6.2.4) the subexpression e' :`y.value` evaluates to a reference into the heap. The type for e' is returned by the type judgement as $(\texttt{Int}, 0/1)$ which determines that the assignment consumes one resource to cover the resource units stored in the type of the variable `y`. For this reason, the return type in the typing judgement is a full type rather than a pre-type.

For consistency and conciseness of the presentation of the typing rules, AmorJiSe also represents the function types $O^+ \times t^+, n \to t^+ n'$ with full types O^+, t^+.

6.3.6.5 Paths

Typed object values represent infinite trees which can be traversed along paths containing field names:

Definition 6.3.17.

1. For a full type t_0 let a *type path* be a finite sequence $m_1, m_2, ..., m_k$ such that $m_i \in \texttt{Dom}([\![t_{i-1}]\!]^t)$ and $[\![t_{i-1}]\!]^t(m_i) = (t_i, \psi_i)$ for all $i = 1, 2, ..., k$. Furthermore, extend the type lookup to $t(p) = (t_k, \psi_k)$ and define $\mathcal{P}(t)$ as the set of all valid paths for the type t.

2. For a typed value $v_0 : t_0$ a *heap path* is a finite sequence of field names $m_1, m_2, ..., m_k$ such that

$$\begin{aligned} m_i &\in \texttt{Dom}(H(v_{i-1})) \cap \texttt{Dom}([\![t_{i-1}]\!]^t) \\ [\![t_{i-1}]\!]^t(m_i) &= (t_i, \psi_i) \text{ for } i = 1, 2, ..., k \\ v_i &= H(v_{i-1})(m_i) \text{ for } i = 1, 2, ..., k. \end{aligned}$$

   Let the notation $v(p) = v_k$ be the extension of the object value lookup operator and $\mathcal{P}_H(v : t)$ the set of all heap paths for the typed value $v : t$ in the heap H.

3. Let the notations $\mathtt{x}.m_1.\cdots.m_k$ or $\mathtt{this}.m_1.\cdots.m_k$ describe the paths $m_1,...,m_k$ starting at the values $\chi(\mathtt{x})$ or $\chi(\mathtt{this})$ and $v.m_1.m_2.\cdots.m_k$ describe a path starting at the result v of the last computation.

4. Given a Heap H, and the typed value $v:t$ resulting from the last computation, the lookup for paths extends the existing maps Γ,χ in the intuitive way:

$$\begin{aligned}\Gamma_t(\text{"}v\text{"}) &= (t,\bullet)\\ \Gamma_t(p.m) &= [\![t_p]\!]^t(m) \qquad \text{where } \Gamma_t(p)=(t_p,\Psi_p)\end{aligned}$$

$$\begin{aligned}\chi_{H,v}(\text{"}v\text{"}) &= v\\ \chi_{H,v}(p.m) &= H(\chi_{H,v}(p))(m)\end{aligned}$$

If the values for H and $v:t$ are unambiguous, the map Γ_t is abbreviated as Γ and $\chi_{H,v}$ is abbreviated as χ.

Definition 6.3.18. Given a path p, an heap H and a typed value $v:t$;

1. The set of *continuing paths* is defined as

$$Reach(p)=\{p' \mid p \text{ is a prefix of } p' \text{ and } \chi_{H,v}(p)\neq \mathcal{U}df\}$$

2. The set of *aliasing paths* is defined as

$$Alias(p)=\{p' \mid \chi_{H,v}(p)=\iota=\chi_{H,v}(p')\}$$

Lemma 6.3.19. For two path $p'\in Alias(p)$ it holds $Reach(p)=Reach(p')$.

Proof. The fact $p'\in Alias(p)$ implies $\chi(p')=\iota=\chi(p)$. The definition of $\chi_{H,v}$ implies that $\chi(p',m_1,...,m_k)=\chi(p,m_1,...,m_k)$ and, in particular, $\chi(p',m_1,...,m_k)\neq\mathcal{U}df$ iff $\chi(p,m_1,...,m_k)\neq\mathcal{U}df$. □

6.3.6.6 Annotation constraints

The typing rules derive a set of constraints on the annotations n and N. The basic constraints are linear constraints of the form

$$n_1+...+n_k\leq n'_1+...+n'_{k'}+c.$$

Additionally, AmorJiSe uses higher level constraints as presented in the following sections. They are used as a concise way to compare the nested annotations inside the compared types with each other. Each of these higher level constraints can be translated into a set of linear constraints containing one annotation from each compared type.

In general these types (e.g. in the case of recursive object types or functions) are infinite trees, where each node includes an annotation. The trees are, however, finitely represented using the recursion operator μ and, therefore, only include finitely many different annotations. Thus, the size of the resulting set of linear constraints resulting from a higher level constraint is bounded by the number of possible combinations of these annotations. Each of these sets is, therefore, finite.

Algorithmically a higher level constraint is translated by traversing the infinite trees while adding the correct linear constraint for each visited note in the tree. If the same set of notes is compared a second time the traversal of this branch of the tree is terminated, since all following notes have already been compared as well.

Sharing

If an operation creates a new alias for a value, AmorJiSe needs to make sure the reserved resource units are shared between the new aliases. The sharing relation $t_1 \hookrightarrow t_2 \oplus t_3$. This relation is of particular need when reading variable types from the context. After the read, the variable is still available in the context, but the value returned by the expression also accesses the same data structure. Therefore, the sum of all resource units stored in the context before the read has to be split into the context type after the read plus the type of the return value.

The sharing relation has the form $t_1 \hookrightarrow t_2 \oplus t_3$ and requires that the tree types t_1, t_2 and t_3 have the same set $\mathcal{P}$ of type paths. A given sharing relation is translated into the set of constraints:

$$C(t_1 \hookrightarrow t_2 \oplus t_3) = \bigcup_{p \in \mathcal{P}} \{[\![t_1(p)]\!]^n = [\![t_2(p)]\!]^n + [\![t_3(p)]\!]^n, [\![t_1(p)]\!]^N = [\![t_2(p)]\!]^N = [\![t_3(p)]\!]^N\}$$

Structured annotation constraints

AmorJiSe introduces the four *structured constraints*

$$t\ {}_n{\leq}_n\ t',\ t\ {}_n{\leq}_N\ t',\ t\ {}_N{\leq}_n\ t' \text{ and } t\ {}_N{\leq}_N\ t'.$$

For given $\eta, \eta' \in \{n, N\}$ the constraint $t\ {}_{\eta}{\leq_{\eta'}}\ t'$ asserts that the annotation $[\![t]\!]^{\eta}$ and all nested η annotations in t are less than the corresponding η' annotations in t'. For example, while writing to the memory, AmorJiSe must ensure that all nested n annotations in the type t' for the new value suffice for the capacity N in the type t of the overwritten memory location. This is expressed as $t\ {}_{N}{\leq_{n}}\ t'$. A given constrain $t\ {}_{\eta}{\leq_{\eta'}}\ t'$ translates into the following linear constraints:

$$\mathcal{C}(t\ {}_{\eta}{\leq_{\eta'}}\ t') = \bigcup_{p \in \mathcal{P}(t')} \{[\![t(p)]\!]^{\eta} \leq [\![t'(p)]\!]^{\eta'}\}$$

The following constraints are defined analogously:

- $t\ {}_{\eta}{\geq_{\eta'}}\ t'$ generates the same constraints as $t'\ {}_{\eta'}{\leq_{\eta}}\ t$.
- $t\ {}_{\eta}{=_{\eta'}}\ t'$ generates the same constraints as $t\ {}_{\eta}{\leq_{\eta'}}\ t'$ and $t\ {}_{\eta}{\geq_{\eta'}}\ t'$.
- The relation $t \leq t'$ generates the constraints as $t\ {}_{n}{\geq_{n}}\ t'$ and $t\ {}_{N}{=_{N}}\ t'$. This ensures that at least as many resources can be read from the subtype as from the supertype and the resources written to the supertype suffice to satisfy the subtypes requirements.
- Equally, $t = t'$ produces all constraints which are produced by $t \leq t'$ and $t' \leq t$.
- Contexts are equivalent to object types and therefore the same constraints can be defined. Specifically, for contexts $\Gamma, \Gamma_1, \Gamma_2$ the constraint $\Gamma\ {}_{\eta}{=_{\eta}}\ \downarrow(\Gamma_1, \Gamma_2)$ generates the constraints $\Gamma\ {}_{\eta}{\leq_{\eta}}\ \Gamma_1$ and $\Gamma\ {}_{\eta}{\leq_{\eta}}\ \Gamma_2$. The constraint $\Gamma\ {}_{\eta}{=_{\eta}}\ \uparrow(\Gamma_1, \Gamma_2)$ is defined equivalently with $\geq$.

Empty types

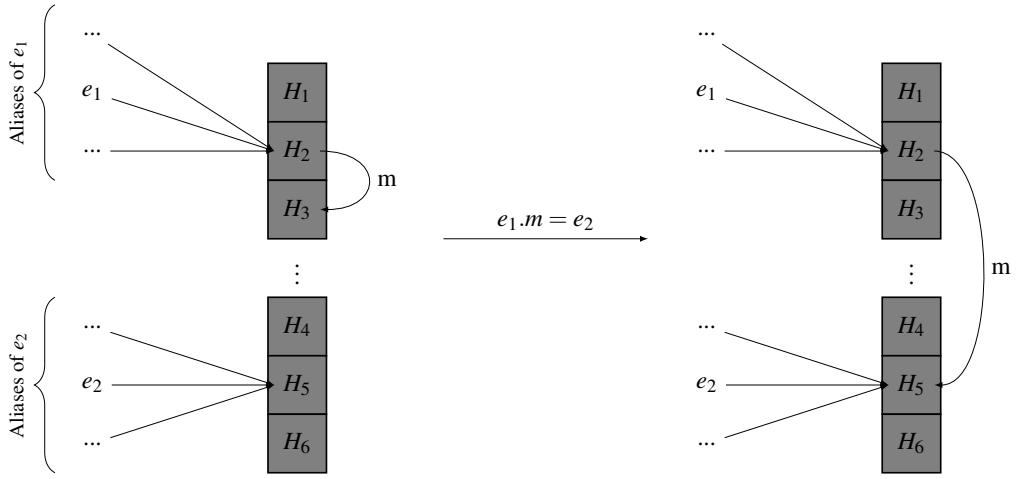
In the rules for writing to objects the constraint t^0 is required, which sets all the n annotations in the type t to 0. This means this type does not contain any reserved resource units. It can be translated into the following set of linear constraints:

$$\mathcal{C}(t^0) = \bigcup_{p \in \mathcal{P}(t)} \{[\![t(p)]\!]^{n} = 0\}$$

6.3.6.7 Typing rules

The typing rules are presented in Figure 6.5. Given a candidate for a full typing, these rules can be used to verify the constraints for the given annotations. For type inference purposes, they can generate linear constraints on the annotations from known data types. A solution for these constraints then constructs a valid full typing.

The constraints in the rules (T-MEMW•) deserve a little extra thought. The rule deals with the assignment to object fields $e_1.m = e_2$ and has to handle a complex set of aliases, as shown in the following heap illustration:



Before the assignment, e_1 and e_2 each have a set of aliases. After the assignment, $e_1.m$ is also an alias of e_2. This imposes the following constraints on the amortised annotations after the assignment:

- The succeeding code might read the value of e_2 through the field m of any alias of e_1. Therefore, the type $\mathcal{T}(e_2)$ for the expression e_2 must provide enough reserved resource units to cover the resource units reserved for all those aliases. This yields the constraint $\mathcal{T}(e_2)\ {}_n{\geq}_N\ t_m$.

- The code might modify the value of e_2 through any alias of e_2. Therefore, the capacity N of all aliases of e_2 has to include the reserved resource units of the aliases of e_1. This is expressed as $\mathcal{T}(e_2)\ {}_N{\geq}_N\ t_m$.

- Writing to the aliases of e_1 needs to provide enough resource units to cover the reserved units in all aliases of e_2. This is expressed by $t_m\ {}_N{\geq}_N\ \mathcal{T}(e_2)$.

Figure 6.5 Typing rules

$$\frac{\mathcal{T}(c) = (t_c, 0/0)}{\Gamma, n + N + c_{(\text{VAL})} \vdash c : (\mathcal{T}(c), N/N) | \Gamma, n} \text{ (T-VAL)} \quad c \in \mathbb{Z} \cup \{\texttt{true}, \texttt{false}\}$$

$$\frac{\begin{array}{c}\Gamma(var) = (\mathcal{T}(var,1), \bullet) \\ \Gamma' = \Gamma[var \mapsto (\mathcal{T}(var,2), \bullet)] \\ \mathcal{T}(var,1) \hookrightarrow \mathcal{T}(var,2) \oplus \mathcal{T}(var,3)\end{array}}{\Gamma, n + c_{(\text{VARR})} \vdash var : \mathcal{T}(var,3) | \Gamma', n} \text{ (T-VARR)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e : \mathcal{T}(e,1) | \Gamma_1, n' \\ \Gamma_1(\mathtt{x}) = (\mathcal{T}(var), \psi_{var}) \\ \Gamma' = \Gamma_1[var \mapsto (\mathcal{T}(var), \bullet)] \\ \mathcal{T}(e,1) \hookrightarrow \mathcal{T}(e,2) \oplus \mathcal{T}(var = e) \\ \mathcal{T}(e,2) \leq \mathcal{T}(var)\end{array}}{\Gamma, n + c_{(\text{VARW})} \vdash var = e : \mathcal{T}(var = e) | \Gamma', n'} \text{ (T-VARW)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e : \mathcal{T}(e) | \Gamma', n' \\ \mathcal{T}(e)(m) = (\mathcal{T}(e.m), \bullet)\end{array}}{\Gamma, n + c_{(\text{MEMR})} \vdash e.m : \mathcal{T}(e.m) | \Gamma', n'} \text{ (T-MEMR)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e : \mathcal{T}(e) | \Gamma_2, n' \\ \Gamma_2(var) = (\mathcal{T}(var,1), \bullet) \\ (var = e') \text{ is not a subexpr. of } e \\ \mathcal{T}(var,1)(m) = (t_m, \psi_m) \\ \Gamma' = \Gamma_2[var \mapsto (\mathcal{T}(var,2), \bullet)] \\ \mathcal{T}(var,2)(m) = (t'_m, \bullet) \\ t'_m = t_m \\ \mathcal{T}(var,2) = \mathcal{T}(var,1) \\ \mathcal{T}(var.m = e)^0 \qquad \mathcal{T}(var.m = e) \ {}_N{=}_N \ \mathcal{T}(e) \\ \mathcal{T}(e) \ {}_N{=}_N \ t_m, \ t_m \ {}_N{=}_N \ t'_m \\ \mathcal{T}(e) \ {}_n{=}_N \ \mathcal{T}(e)\end{array}}{\Gamma, n + c_{(\text{MEMW})}(\psi_m = \bullet) + c_{(\text{VARR})} \vdash var.m = e : \mathcal{T}(var.m = e) | \Gamma', n'} \text{ (T-MEMW}\circ\text{)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e_1 : \mathcal{T}(e_1) | \Gamma_1, n' \\ \Gamma_1, n' \vdash e_2 : \mathcal{T}(e_2) | \Gamma', n'' \\ - \\ \mathcal{T}(e_1)(m) = (t_m, \bullet) \\ - \\ - \\ - \\ - \\ \mathcal{T}(e_1.m = e_2)^0 \\ \mathcal{T}(e_2) \ {}_N{=}_N \ t_m \\ \mathcal{T}(e_2) \ {}_n{=}_N \ \mathcal{T}(e_2)\end{array}}{\Gamma, n + c_{(\text{VARW})}(true) \vdash e_1.m = e_2 : \mathcal{T}(e_1.m = e_2) | \Gamma', n''} \text{ (T-MEMW}\bullet\text{)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e_b : \mathcal{T}(e_b) | \Gamma_1, n' \\ \Gamma_1, n' \vdash e_t : \mathcal{T}(e_t) | \Gamma_t, n_t \\ \Gamma_1, n' \vdash e_f : \mathcal{T}(e_f) | \Gamma_f, n_f \\ \mathcal{T}(e_t) \leq \mathcal{T}(e_b ? e_t : e_f) \qquad \mathcal{T}(e_f) \leq \mathcal{T}(e_b ? e_t : e_f) \\ \Gamma' \ {}_N{=}_N \uparrow(\Gamma_f, \Gamma_t) \qquad \Gamma' \ {}_n{=}_n \downarrow(\Gamma_t, \Gamma_f) \\ n'' = min(n_t, n_f)\end{array}}{\Gamma, n + c_{(\text{COND})} \vdash e_b ? e_t : e_f : \mathcal{T}(e_b ? e_t : e_f) | \Gamma', n''} \text{ (T-COND)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e_1 : \mathcal{T}(e_1) | \Gamma_1, n' \\ \Gamma_1, n' \vdash e_2 : \mathcal{T}(e_2) | \Gamma', n'' \\ \mathcal{T}(e_1; e_2) = \mathcal{T}(e_2)\end{array}}{\Gamma, n + c_{(\text{SEQ})} \vdash e_1; e_2 : \mathcal{T}(e_1; e_2) | \Gamma', n''} \text{ (T-SEQ)}$$

The rule (T-MEMW∘) combines the last two constraints to $\mathcal{T}(e_2)\,{}_N{=}_N\,t_m$. Furthermore, these constraints result in the following chain:

$$\mathcal{T}(e_2)\,{}_n{\leq}_N\,\mathcal{T}(e_2)\,{}_N{=}_N\,t_m\,{}_N{\leq}_n\,\mathcal{T}(e_2)$$

which can only be true if $\mathcal{T}(e_2)\,{}_n{=}_N\,\mathcal{T}(e_2)\,{}_N{=}_N\,t_m$. In turn, this means all reserved resource units in the type $\mathcal{T}(e_2)$ are now only available through the aliases of e_1. Therefore, the value returned by the expression $e_1.m = e_2$ cannot contain any reserved resource units anymore, which is expressed by $\mathcal{T}(e_1.m = e_2)^0$.

The equivalent consideration leads to the constraints in (T-MEMW∘) in addition to the constraints for the update of the type for the field m.

6.4 Properties

The following section discusses the formal properties of AmorJiSe with the goal of proving the soundness theorem. Objects are of the greatest interest, as they represent the recursive data structures and are unique to JavaScript. Without loss of generality, the following definitions assume that for a typed object $\{m_1 : v_1, ..., m_k : v_k, m_{k+1} : v_{k+1}, ...\} : \mu\alpha.[m_1 : (t_1, \psi_1), ..., m_k : (t_k, \psi_k), m'_1 : (t'_1, \circ), ...]$ the shared fields $m_1, ..., m_k$ are mentioned in the same order in the value and the type. With the additional disjunct fields $\{m_{i+1}, ...\} \cap \{m'_1, ...\} = \{\}$, the value contains fields $m_{k+1}, ...$ not described by the type and the type mentions fields $m'_1, ...$ marked as $\circ$, which are not contained in the value.

6.4.1 Definitions

The upcoming soundness statement for AmorJiSe (Theorem 6.4.10) intuitively states that, for every expression e, the resources reserved for the expression via the inferred types are enough to execute e. Formally this number of available resource units is defined as *potential*. It depends on the annotations inside the types and the structure of the typed value.

Definition 6.4.1. 1. Given a typed value $v : t$ in the heap H, the total resource *potential* $\Sigma_H v : t$ is defined as:

$$\Sigma_H v : t = \sum_{p \in \mathcal{P}_H(v:t)} [\![t(p)]\!]^n$$

2. In extension, the potential of a whole environment (Γ, H, χ) is defined as:

$$\Sigma_H \chi : \Gamma = \sum_{\mathtt{x} \in \mathtt{Dom}(\Gamma) \cap \mathtt{Dom}(\chi)} \Sigma_H \chi(\mathtt{x}) : \Gamma(\mathtt{x})$$

The analysis discusses two different kinds of states: pre-states (H, χ, Γ, n) are the inputs for the typing/evaluation relation and post-states $(H, \chi, \Gamma, n, v : t)$ are the output of the relations. The potential of either of those is defined as the sum of the potential of its parts:

$$\begin{aligned} \mathcal{N}(H, \chi, \Gamma, n) &= \Sigma_H \chi : \Gamma + n \\ \mathcal{N}(H, \chi, \Gamma, n, v : t) &= \Sigma_H \chi : \Gamma + n + \Sigma_H v : t \end{aligned}$$

Lemma 6.4.2. Due to the definition and the domain $\mathbb{N}$ for the annotations n, the potential $\Sigma_H v : t$ is always non-negative. This property extends to $\Sigma_H \chi : \Gamma$, $\mathcal{N}(H, \chi, \Gamma, n)$ and $\mathcal{N}(H, \chi, \Gamma, n, v : t)$.

Definition 6.4.3.

1. A type t is called *sufficient for* the set of types $\{t_1, ..., t_k\}$, if for all type paths $p \in \mathcal{P}(t)$ it holds: $[\![t(p)]\!]^N \geq \sum_{i=1...k} [\![t_i(p)]\!]^n$.
2. Given a state H, χ, Γ, potentially a typed value $v : t$ and a heap path p call the type $\Gamma(p)$ sufficient, if $\Gamma(p)$ is sufficient for the set of all alias types $\{\Gamma(p') | p' \in Alias(p)\}$. This is written as $\Gamma, H, \chi \vdash \Gamma(p)\checkmark$.
3. Furthermore, the context Γ is called sufficient for H, χ and potentially $v : t$, if for all heap paths p the type $\Gamma(p)$ is sufficient. This is denoted as $\Gamma, H, \chi \vdash \Gamma\checkmark$

Intuitively, the capacity of a sufficient type is at least as big as the sum of all reserved units in the alias types. Therefore, it fulfils the desired property of the capacity N.

Lemma 6.4.4. If $\Gamma(p)$ is sufficient, then for every $p' \in Reach(p)$ the type $\Gamma(p')$ is also sufficient.

Proof. This follows from Definition 6.4.3, since the set $\mathcal{P}(\Gamma(p'))$ is only a sub-set of $\mathcal{P}(\Gamma(p))$. □

Remark 6.4.5. There are two things to notice about this notation:

1. If two paths p and p' are aliased, the fact that Γ is sufficient requires both paths p and p' to be sufficient.

2. The notion of sufficient does not depend on the field state of the fields in the object types. As shown in example 6.2.4 (page 121), if an object is extended via one alias all other aliases need be be up-to-date with the capacity of this new field. Since AmorJiSe does not track all possible aliases of all values, the information about the capacity needs to be synchronised between the aliases during alias creation. This is achieved via the capacity of potential ($\circ$) fields which in turn is taken into consideration for the definition of sufficient.

To remain sufficient while creating new aliases it is important that the splitting relation does indeed split the potential between the two output types:

Lemma 6.4.6. For full types t, t_1, t_2 and a value v in a given heap H, if $t \hookrightarrow t_1 \oplus t_2$ then $\Sigma_H v : t = \Sigma_H v : t_1 + \Sigma_H v : t_2$.

Proof. Remember that the sharing relation requires $\mathcal{P}(t) = \mathcal{P}(t_1) = \mathcal{P}(t_2)$. The same relation between the heap paths $\mathcal{P}_H(v : t) = \mathcal{P}_H(v : t_1) = \mathcal{P}_H(v : t_2)$ follows from the definition of heap paths (Definition 6.3.17).

$$
\begin{aligned}
\Sigma_H v : t &= \sum_{p \in \mathcal{P}_H(v:t)} [\![t(p)]\!]^n \\
&\overset{sharing}{=} \sum_{p \in \mathcal{P}_H(v:t)} ([\![t_1(p)]\!]^n + [\![t_2(p)]\!]^n) \\
&= \sum_{p \in \mathcal{P}_H(v:t)} [\![t_1(p)]\!]^n + \sum_{p \in \mathcal{P}_H(v:t)} [\![t_2(p)]\!]^n \\
&= \sum_{p \in \mathcal{P}_H(v:t_1)} [\![t_1(p)]\!]^n + \sum_{p \in \mathcal{P}_H(v:t_2)} [\![t_2(p)]\!]^n \\
&= \Sigma_H v : t_1 + \Sigma_H v : t_2
\end{aligned}
$$

□

6.4.2 Heap loops

One interesting case is the potential of loops in data-structures in the heap. If an object value o stored at a heap address ι has a reference to ι directly as the value of one of its members or indirectly through multiple members, this heap contains a loop. If such a loop is typed with an annotated recursive object type O, there exists infinite many heap paths for the object o. The potential of this value sums up over all those paths and sums up the annotations contained in O infinitely often. If one of these annotations is positive, the potential becomes infinite itself.

If the heap loop is provided as the input heap configuration, the analysis correctly infers values for all annotations in O and the following soundness result states that the resource requirement is the potential of the input values. If one of the annotations contained in the type of the loop is positive this potential is infinite. The result is therefore trivial: "The expression executes with at most infinite resource units". In the other case, all annotations inside the type of the loop are 0, the potential of the loop is 0 itself and irrelevant for the resource consumption.

The situation is more complicated if a heap loop gets created during runtime. For example, if an object is assigned its own heap address as one of the fields. The following lemma guarantees that such loops typed with a sufficient type have a potential of 0 under certain conditions, which will be fulfilled in the proof later.

Lemma 6.4.7. Assume a state H, χ, Γ and an address ι with $H(\iota) \neq \mathcal{U}df$. Let the address $v = \iota$ as value be typed with the sufficient type t with $t\ {}_N{=}_n\ t$ and let there be a non trivial path $p_\iota \in \mathcal{P}_H(v)$ with $\chi_H(p_\iota) = \iota$. Then, for any path $p \in Reach(\iota)$ and any $p' \in Alias(p)$ it holds $[\![\Gamma(p')]\!]^n = 0$. This implies $\sum_{p' \in Alias(\iota)} [\![\Gamma(p', p'')]\!]^n = 0$ and $\Sigma_H v : t = 0$.

$\Sigma_H v : t = 0$ and $\sum_{p' \in Alias(\iota)} [\![\Gamma(p', p'')]\!]^n = 0$ for any concatenated path p', p''.

Proof. Choose a path $p \in Reach(\iota)$. By the assumptions, the concatenated path $p_1 = p_\iota, p$ is an alias to p itself $p \in Alias(p_1)$.

The definition of "t is sufficient" (Definition 6.4.3) requires

$$[\![t(p_1)]\!]^N \geq \sum_{p' \in Alias(p_1)} [\![t(p')]\!]^n.$$

Due to $t\ {}_n{=}_N\ t$ it holds $[\![t(p_1)]\!]^N = [\![t(p_1)]\!]^n$ and due to $p_1 \in Alias(p_1)$, $[\![t(p_1)]\!]^n$ can be subtracted from both sides of this relation, resulting in

$$0 \geq \sum_{p' \in Alias(p_1) - p_1} [\![t(p')]\!]^n.$$

Since all $[\![t(p')]\!]^n \geq 0$ this is equivalent with $[\![t(p')]\!]^n = 0$ for all $p' \in Alias(p_1) - p_1$. The same construction can be repeated for $p_2 = p_\iota \cdot p_1$, proving $[\![t(p_1)]\!]^n = 0$.

The other two claims follow, since the claimed terms are composed of summands of the form $[\![t(p')]\!]^n = 0$. □

6.4.3 Soundness

The soundness of AmorJiSe relates the evaluation and the typing relation. To prove soundness, AmorJiSe assumes the soundness of the underlying system, which expresses that the type of a typed object $v : O$ describe the object value v correctly. This is expressed by the agreement relation:

Definition 6.4.8. Given a state H, χ, Γ, a value v agrees with its type t, written as $H, \chi, \Gamma \vdash v : t$, if for all paths p from v:

- $t(p) = Int \Rightarrow \chi_H(v) \in Int$
- $t(p) = Bool \Rightarrow v(p) \in Bool$
- $t(p) = \mu\alpha.[m_1 : (t_1, \psi_1), ..., m_k : (t_k, \psi_k)] \Rightarrow \forall i = 1...k$

$$\psi_i = \bullet \Rightarrow m_i \in \mathtt{Dom}(H(v(p)))$$

Furthermore, in a given heap H a stack χ agrees with the context Γ, written as $H, \chi, \Gamma \vdash$, if for all variables $var \in \mathtt{Dom}(\Gamma)$

$$\Gamma(var) = (t, \bullet) \Rightarrow var \in \mathtt{Dom}(\chi) \wedge H, \chi, \Gamma \vdash \chi(var) : \Gamma(var)$$
$$\Gamma(var) = (t, \circ) \wedge var \in \mathtt{Dom}(\chi) \Rightarrow H, \chi, \Gamma \vdash \chi(var) : \Gamma(var)$$

Theorem 6.4.9. *(Assumption) Given an expression e, which can be typed and evaluated in a matching heap H, stack χ and context Γ*

$$e, H, \chi \xrightarrow[n'_S]{n_S} \!\!\!\!\rightarrow v, H', \chi'$$
$$\Gamma, n_T \vdash e : t | \Gamma', n'_T$$
$$H, \chi, \Gamma \vdash$$

Then the value v can be typed as t

$$H, \chi, \Gamma \vdash v : t$$

and the post-state agrees

$$H', \chi', \Gamma' \vdash .$$

In the soundness theorem for AmorJiSe this is extended by the properties of the annotations:

Theorem 6.4.10. *Given an expression e, which can be typed and evaluated in a matching heap H, stack χ and a sufficient context Γ*

$$\begin{gathered} e, H, \chi \xrightarrow[n'_S]{n_S} \twoheadrightarrow v, H', \chi' \\ \Gamma, n_T \vdash e : t | \Gamma', n'_T \\ H, \chi, \Gamma \vdash \\ H, \chi, \Gamma \vdash \Gamma\checkmark \end{gathered}$$

the following holds:
The potential in the pre-state $\mathcal{N}(H, \chi, \Gamma, n_T)$ is enough to execute e:

$$\mathcal{N}(H, \chi, \Gamma, n_T) \geq n_S.$$

The potential is reduced by at least as much as the execution of e consumed resource units:

$$\mathcal{N}(H, \chi, \Gamma, n_T) - \mathcal{N}(H', \chi', \Gamma', n_T, v : t) \geq n_S - n'_S$$

and the context Γ' is sufficient for the heap H' as well as t for the value v.

$$\begin{gathered} H', \chi', \Gamma' \vdash \Gamma'\checkmark \\ H', \chi', \Gamma' \vdash v : t\checkmark. \end{gathered}$$

Proof. This claim is proven by induction on the length of the derivation of the evaluation relation $e, H, \chi \xrightarrow[n'_S]{n_S} \twoheadrightarrow v, H', \chi'$. Since all rules of both the evaluation relation and the type checking relation are syntax-driven, the induction step distinguishes one case for each evaluation rule. In each case, the handled evaluation rule matches exactly one typing rule, except for the rule (T-MEMW). In this exception the proof is performed in the two sub-cases for the typing rules (T-MEMW$\circ$) and (T-MEMW$\bullet$).

Case (S-VAL): In this basic case it holds $n_S = c_{(\text{VAL})}, n'_S = 0$ as well as $n_t = n + N + c_{(\text{VAL})}, n'_t = n$. That directly results in

$$\mathcal{N}(H, \chi, \Gamma, n_T) \geq c_{(\text{VAL})} = n_S$$

and

$$\begin{aligned}\mathcal{N}(H,\chi,\Gamma,n_T) &= \Sigma_H\chi:\Gamma + (n+N+c_{(\text{VAL})})\\ &= \Sigma_H\chi:\Gamma + n + c_{(\text{VAL})} + \Sigma_H c:(T(c),N/N)\\ &= \mathcal{N}(H,\chi,\Gamma,n'_T,c:(T(c),N/N)) + c_{(\text{VAL})}\end{aligned}$$

implies

$$\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H,\chi,\Gamma,n'_T,c:(T(c),N/N)) = c_{(\text{VAL})} = n_S - n'_S$$

The resulting environment H,χ,Γ is sufficient due to the precondition. The only path reachable from the path "v" is "v" and since the value c is not stored in the heap H, it does not have any other aliasing paths. Therefore, $H,\chi,\Gamma \vdash v : t\checkmark$ follows from $[\![\Gamma(\text{"}v\text{"})]\!]^n = N \leq N = [\![\Gamma(\text{"}v\text{"})]\!]^N$.

Case (S-VARR): In this case there are 2 cases for $var \in \{\texttt{x},\texttt{this}\}$. Without loss of generality, the following assumes $var = \texttt{x}$. The rules in this case provide the values $n_S = c_{(\text{VARR})}, n'_S = 0$ and $n_T = n + c_{(\text{VARR})} \geq c_{(\text{VARR})}$. The claim

$$\mathcal{N}(H,\chi,\Gamma,n_T) \geq n_T \geq c_{(\text{VARR})} = n_S$$

holds immediately. Due to Lemma 6.4.6 and $\mathcal{T}(\texttt{x},1) \hookrightarrow \mathcal{T}(\texttt{x},2) \oplus \mathcal{T}(\texttt{x},3)$, it holds

$$\Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},1) = \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},2) + \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},3)$$

This implies

$$\begin{aligned}\Sigma_H\chi:\Gamma &= \Sigma_H\chi(\texttt{this}):\Gamma(\texttt{this}) + \Sigma_H\chi(\texttt{x}):\Gamma(\texttt{x})\\ &= \Sigma_H\chi(\texttt{this}):\Gamma'(\texttt{this}) + \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},1)\\ &= \Sigma_H\chi(\texttt{this}):\Gamma'(\texttt{this}) + \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},1)\\ &= \Sigma_H\chi(\texttt{this}):\Gamma'(\texttt{this}) + \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},2) + \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},3)\\ &= \Sigma_H\chi(\texttt{this}):\Gamma'(\texttt{this}) + \Sigma_H\chi(\texttt{x}):\Gamma'(\texttt{x}) + \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},3)\\ &= \Sigma_H\chi:\Gamma' + \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},3)\end{aligned}$$

and leads to

$$\begin{aligned}&\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H,\chi,\Gamma',n'_T,\chi(\texttt{x}):\mathcal{T}(\texttt{x},3))\\ =&\mathcal{N}(H,\chi,\Gamma,n_T) - \Sigma_H\chi:\Gamma' - n'_T - \Sigma_H\chi(\texttt{x}):\mathcal{T}(\texttt{x},3)\\ =&\mathcal{N}(H,\chi,\Gamma,n_T) - \Sigma_H\chi:\Gamma - n'_T\\ =&\Sigma_H\chi:\Gamma + n_T - \Sigma_H\chi:\Gamma - n'_T\\ =&c_{(\text{VARR})} = n_S - n'_S\end{aligned}$$

It remains to show the sufficiencies. By the precondition $H,\chi,\Gamma \vdash \Gamma\checkmark$ all paths are sufficient in the previous environment. In the new environment the type $\Gamma'(\mathtt{x})$ was changed from $\mathcal{T}(\mathtt{x},1)$ to $\mathcal{T}(\mathtt{x},2)$. Since they are related via the sharing relation, $\mathcal{T}(\mathtt{x},1)$ and $\mathcal{T}(\mathtt{x},2)$ have the same structure and are equal in all nested capacities N. Now consider a path p in the new environment H,χ,Γ' which does not start with v. Therefore, p exists in the old environment H,χ,Γ and its set of alias paths in this environment is $P = \{p' | H(p') = H(p)\}$. This set splits into

$$P_{\mathtt{x}} = \{\mathtt{x}.p' | H(\mathtt{x}.p') = H(p)\} \text{ and}$$
$$P_{\mathtt{this}} = \{\mathtt{this}.p' | H(\mathtt{this}.p') = H(p)\}.$$

Since v is a new alias of $\mathtt{x}$, in the new environment H,χ,Γ' the path p has the new set of alias paths $P' = P \cup \{v.p' | \mathtt{x}.p' \in P_{\mathtt{x}}\}$. For each $\mathtt{x}.p' \in P'$ by the definition of the sharing relation it holds $[\![\Gamma(\mathtt{x}.p')]\!]^n = [\![\Gamma'(\mathtt{x}.p')]\!]^n + [\![\Gamma'(v.p')]\!]^n$ which results in:

$$\begin{aligned}
[\![\Gamma'(p)]\!]^N &= [\![\Gamma(p)]\!]^N \\
&\geq \sum_{p' \in P} [\![\Gamma(p')]\!]^n \\
&= \sum_{p' \in P_{\mathtt{this}}} [\![\Gamma(p')]\!]^n + \sum_{\mathtt{x}.p' \in P_{\mathtt{x}}} [\![\Gamma(\mathtt{x}.p')]\!]^n \\
&= \sum_{p' \in P_{\mathtt{this}}} [\![\Gamma'(p')]\!]^n + \sum_{\mathtt{x}.p' \in P_{\mathtt{x}}} [\![\Gamma'(\mathtt{x}.p')]\!]^n + \sum_{\mathtt{x}.p' \in P_{\mathtt{x}}} [\![\Gamma'(v.p')]\!]^n \\
&= \sum_{p' \in P_{\mathtt{x}}} [\![\Gamma'(p')]\!]^n
\end{aligned}$$

This constitutes the sufficiency of $\Gamma(p)$ and $H,\chi,\Gamma' \vdash \Gamma'\checkmark$ as a whole. Since $[\![\Gamma'(v.p)]\!]^N = [\![\Gamma'(\mathtt{x}.p)]\!]^N$ the sufficiency $H,\chi,\Gamma' \vdash v : t\checkmark$ is equivalent.

Case (S-VARW): Without loss of generality, this case considers $var = \mathtt{x}$. The induction hypothesis on the subexpression e yields

$$\mathcal{N}(H,\chi,\Gamma,n) \geq n_e \tag{6.1}$$
$$\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H',\chi_1,\Gamma_1,n',v:\mathcal{T}(e,1)) \geq n_e - n'_e \tag{6.2}$$

which results in

$$\begin{aligned}
\mathcal{N}(H,\chi,\Gamma,n_T) &= \mathcal{N}(H,\chi,\Gamma,n + c_{(\text{VARW})}) \\
&\overset{6.1}{\geq} n_e + c_{(\text{VARW})} \\
&= n_S.
\end{aligned}$$

Furthermore, the sharing relation $\mathcal{T}(e,1) \hookrightarrow \mathcal{T}(e,2) \oplus \mathcal{T}(\mathtt{x} = e)$ implies

$$\Sigma_H v : \mathcal{T}(e,1) = \Sigma_H v : \mathcal{T}(e,2) + \Sigma_H v : \mathcal{T}(\mathtt{x} = e) \tag{6.3}$$

and by the subtyping $\mathcal{T}(e,2) \leq \mathcal{T}(\mathtt{x})$ it holds

$$\Sigma_{H'} v : \mathcal{T}(e,2) \geq \Sigma_{H'} v : \mathcal{T}(\mathtt{x}) \tag{6.4}$$

This combines to

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H',\chi',\Gamma',n'_T, v : \mathcal{T}(\mathtt{x} = e)) \\
=&\mathcal{N}(H,\chi,\Gamma,n + c_{(\mathrm{VARW})}) - \mathcal{N}(H',\chi',\Gamma',n', v : \mathcal{T}(\mathtt{x} = e)) \\
=&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{VARW})} - \Sigma_{H'}\chi' : \Gamma' - \Sigma_{H'} v : \mathcal{T}(\mathtt{x} = e) - n' \\
=&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{VARW})} - \Sigma_{H'}\chi'(\mathtt{this}) : \Gamma'(\mathtt{this}) - \Sigma_{H'}\chi'(\mathtt{x}) : \Gamma'(\mathtt{x}) \\
&\qquad - \Sigma_{H'} v : \mathcal{T}(\mathtt{x} = e) - n' \\
=&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{VARW})} - \Sigma_{H'}\chi_1(\mathtt{this}) : \Gamma_1(\mathtt{this}) - \Sigma_{H'} v : \mathcal{T}(\mathtt{x}) \\
&\qquad - \Sigma_{H'} v : \mathcal{T}(\mathtt{x} = e) - n' \\
\overset{6.4}{\geq}&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{VARW})} - \Sigma_{H'}\chi_1(\mathtt{this}) : \Gamma_1(\mathtt{this}) - \Sigma_{H'} v : \mathcal{T}(e,2) \\
&\qquad - \Sigma_{H'} v : \mathcal{T}(\mathtt{x} = e) - n' \\
\overset{6.3}{=}&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{VARW})} - \Sigma_{H'}\chi_1(\mathtt{this}) : \Gamma_1(\mathtt{this}) - \Sigma_{H'} v : \mathcal{T}(e,1) - n' \\
\geq&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{VARW})} - \Sigma_{H'}\chi_1 : \Gamma_1 - \Sigma_{H'} v : \mathcal{T}(e,1) - n' \\
\geq&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{VARW})} - \mathcal{N}(H',\chi_1,\Gamma_1,n', v : \mathcal{T}(e,1)) \\
\overset{6.2}{\geq}&n_S - n'_S
\end{aligned}$$

The induction hypothesis guarantees that all types in the old state $H',\chi_1,\Gamma_1, v : \mathcal{T}(e,1)$ are sufficient. Now consider a path p in the new state $H',\chi',\Gamma', v : \mathcal{T}(x = e)$. In general, $Alias(p)$ contains a number of paths $v.m_1.\cdots.m_k$ in the new state. For each such path it also contains the path $\mathtt{x}.m_1.\cdots.m_k$ in the new state. All other alias paths are as in the old state with the same type and value. The path $v.m_1.\cdots.m_k$ is also a valid path in the old state. By the sharing relation, it is clear that the potential of $v.m_1.\cdots.m_k$ in the old state is the same as the summed up potential of the paths $v.m_1.\cdots.m_k$ and $\mathtt{x}.m_1.\cdots.m_k$ in the new state.

Now consider two cases: Either the value of the path p was equal in the old state, then the sum of potential of all aliases is the same in old and new state (with the

split of the potential of all paths $v.\cdots$), or the value of the path p has changed by the assignment. Then p is clearly of the form $\mathrm{x}.m_1.\cdots.m_k$, since no other path was changed. In that case, due to the assignment, the path p has the same alias paths as $v.m_1.\cdots.m_k$ and due to $\mathcal{T}(e,2) \leq \mathcal{T}(x)$ also the same capacity. Since $v.m_1.\cdots.m_k$ has already been established as sufficient, p has to be sufficient, too.

Case (S-MEMR): The induction hypothesis in this case yields:

$$\begin{aligned} \mathcal{N}(H,\chi,\Gamma,n) &\geq n_e \\ \mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H',\chi',\Gamma',n',\mathfrak{l}:\mathcal{T}(e)) &\geq n_e - n'_e \end{aligned}$$

Furthermore, with the auxiliary type $(t_{e.m},\bullet) = \mathcal{T}(e)(m)$, by the definition of the potential it holds

$$\begin{aligned} \mathcal{N}(H',\chi',\Gamma',n',\mathfrak{l}:\mathcal{T}(e)) =& \mathcal{N}(H',\chi',\Gamma',n',H'(\mathfrak{l}):\mathcal{T}(e))) \\ \geq& \mathcal{N}(H',\chi',\Gamma',n',H'(\mathfrak{l})(m):t_{e.m}) \end{aligned}$$

This yields the claimed inequalities:

$$\begin{aligned} &\mathcal{N}(H,\chi,\Gamma,n_T + c_{(\mathrm{MEMR})}) \\ =&\mathcal{N}(H,\chi,\Gamma,n_T) + c_{(\mathrm{MEMR})} \\ \overset{IH}{\geq}& n_e + c_{(\mathrm{MEMR})} = n_S \end{aligned}$$

$$\begin{aligned} &\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H',\chi',\Gamma',n'_T,H'(\mathfrak{l})(m):\mathcal{T}(e.m)) \\ =&\mathcal{N}(H,\chi,\Gamma,n + c_{(\mathrm{MEMR})}) - \mathcal{N}(H',\chi',\Gamma',n',H'(\mathfrak{l})(m):t_{e.m}) \\ =&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{MEMR})} - \mathcal{N}(H',\chi',\Gamma',n',H'(\mathfrak{l})(m):t_{e.m}) \\ \geq&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{MEMR})} - \mathcal{N}(H',\chi',\Gamma',n',\mathfrak{l}:\mathcal{T}(e)) \\ \overset{IH}{\geq}& n_e + c_{(\mathrm{MEMR})} - n'_e = n_S - n'_S \end{aligned}$$

The sufficiency $H',\chi',\Gamma' \vdash \Gamma'\checkmark$ is given by the induction hypothesis and $H',\chi',\Gamma' \vdash H'(\mathfrak{l})(m) : \mathcal{T}(e)(m)\checkmark$ follows from $H',\chi',\Gamma' \vdash \mathfrak{l} : \mathcal{T}(e)\checkmark$ since the paths from $v = H'(\mathfrak{l})(m)$ are equivalent to the subset $Reach(v.m)$ of the paths from $\mathfrak{l} : \mathcal{T}(e)$.

Case (S-SEQ): In this case the induction hypothesis applies directly to the subexpression e_1. One result of this hypothesis is $H_1,\chi_1,\Gamma_1 \vdash \Gamma_1\checkmark$ and thus the induction hypothesis applies for e_2. Therefore, $H',\chi',\Gamma' \vdash \Gamma'\checkmark$ and $H',\chi',\Gamma' \vdash v_2 : \mathcal{T}(e_2)\checkmark$ fol-

lows by induction immediately and the following inequalities can be assumed:

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n) && \geq n_1\\
&\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H_1,\chi_1,\Gamma_1,n',v_1:\mathcal{T}(e_1)) && \geq n_1 - n_1'\\
&\mathcal{N}(H_1,\chi_1,\Gamma_1,n') && \geq n_2\\
&\mathcal{N}(H_1,\chi_1,\Gamma_1,n') - \mathcal{N}(H',\chi',\Gamma',n'',v_2:\mathcal{T}(e_2)) && \geq n_2 - n_2'
\end{aligned}$$

For $n_\wr$ there are the following 2 cases:

Case $n_1' \geq n_2$: It holds $n_\wr = n_1$ and from the induction hypothesis it follows $\mathcal{N}(H,\chi,\Gamma,n) \geq n_1 = n_\wr$.

Case $n_1' < n_2$: It holds $n_\wr = n_1 - n_1' + n_2$. Furthermore, due to the definition of potential, the potential of $v_1 : \mathcal{T}(e_1)$ is non-negative. From that it follows:

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n)\\
=&\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H_1,\chi_1,\Gamma_1,n') + \mathcal{N}(H_1,\chi_1,\Gamma_1,n')\\
\geq&\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H_1,\chi_1,\Gamma_1,n',v_1:\mathcal{T}(e_1)) + \mathcal{N}(H_1,\chi_1,\Gamma_1,n')\\
\geq&n_1 - n_1' + n_2 = n_\wr
\end{aligned}$$

In both cases $\Sigma_H v_1 : \mathcal{T}(e_1) \geq 0$ and therefore

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n_T)\\
=&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\text{SEQ})}\\
\geq&n_\wr + c_{(\text{SEQ})} = n_s
\end{aligned}$$

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H',\chi',\Gamma',n_T',v_2:\mathcal{T}(e_1;e_2))\\
=&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\text{SEQ})} - \mathcal{N}(H',\chi',\Gamma',n'',v_2:\mathcal{T}(e_2))\\
\geq&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\text{SEQ})} - \Sigma_H v_1 : \mathcal{T}(e_1) - \mathcal{N}(H',\chi',\Gamma',n'',v_2:\mathcal{T}(e_2))\\
=&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\text{SEQ})} - \mathcal{N}(H_1,\chi_1,\Gamma_1,n',v_1:\mathcal{T}(e_1))\\
&\qquad + \mathcal{N}(H_1,\chi_1,\Gamma_1,n') - \mathcal{N}(H',\chi',\Gamma',n'',v_2:\mathcal{T}(e_2))\\
\overset{IV}{\geq}&n_1 - n_1' + n_2 - n_2' + c_{(\text{SEQ})}\\
=&n_\wr - n_\wr' + c_{(\text{SEQ})} = n_S - n_S'
\end{aligned}$$

Case (S-CONDTRUE): By the induction hypothesis, H',χ',Γ' is sufficient. There-

fore, the induction hypothesis applies to e_t resulting in the following bounds:

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n) && \geq n'\\
&\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H_1,\chi_1,\Gamma_1,n',\texttt{true}:\mathcal{T}(e_b)) && \geq n_1 - n_1'\\
&\mathcal{N}(H_1,\chi_1,\Gamma_1,n') && \geq n_2\\
&\mathcal{N}(H_1,\chi_1,\Gamma_1,n') - \mathcal{N}(H',\chi',\Gamma_t,n_t,v:\mathcal{T}(e_t)) && \geq n_2 - n_2'
\end{aligned}$$

As $\Sigma_H \texttt{true}:\mathcal{T}(e_b) = 0$ it holds that

$$\mathcal{N}(H_1,\chi_1,\Gamma_1,n',\texttt{true}:\mathcal{T}(e_b)) = \mathcal{N}(H_1,\chi_1,\Gamma_1,n').$$

With those preconditions the same derivation as in the case S-SEQ derives

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n) && \geq n_\wr\\
&\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H',\chi',\Gamma_t,n_t,v:\mathcal{T}(e_t)) && \geq n_\wr - n_\wr'
\end{aligned}$$

It holds $n' \leq n_t$ as it is computed as $n' = min(n_t, n_f)$. From $\Gamma' \,{}_n{=}_n \downarrow(\Gamma_t,\Gamma_f)$ follows $\Sigma_{H'}\chi' : \Gamma' \leq \Sigma_{H'}\chi' : \Gamma_t$ and from $\mathcal{T}(e_t) \leq \mathcal{T}(e_b ? e_t : e_f)$ follows $\Sigma_{H'} v : \mathcal{T}(e_b ? e_t : e_f) \leq \Sigma_{H'} v : \mathcal{T}(e_t)$. Combining this yields

$$\begin{aligned}
&\mathcal{N}(H',\chi',\Gamma',n',v:\mathcal{T}(e_b ? e_t : e_f))\\
=&\Sigma_{H'}\chi' : \Gamma' + \Sigma_{H'} v : \mathcal{T}(e_b ? e_t : e_f) + n'\\
\leq&\Sigma_{H'}\chi' : \Gamma_t + \Sigma_{H'} v : \mathcal{T}(e_t) + n_t\\
=&\mathcal{N}(H',\chi',\Gamma_t,n_t,v:\mathcal{T}(e_t))
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H',\chi',\Gamma',n_T,v:\mathcal{T}(e_b ? e_t : e_f))\\
=&\mathcal{N}(H,\chi,\Gamma,n + c_{(\text{COND})}) - \mathcal{N}(H',\chi',\Gamma',n',v:\mathcal{T}(e_b ? e_t : e_f))\\
\geq&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\text{COND})} - \mathcal{N}(H',\chi',\Gamma_t,n_t,v:\mathcal{T}(e_t))\\
\geq&n_\wr + c_{(\text{COND})} - n_\wr = n_S - n_S'
\end{aligned}$$

The induction hypothesis also ensures that H',χ',Γ_t are sufficient. Due to the requirements $\Gamma' \,{}_N{=}_N \uparrow(\Gamma_f,\Gamma_t)$ and $\Gamma' \,{}_n{=}_n \downarrow(\Gamma_t,\Gamma_f)$ this immediately proves $[\![\Gamma_t(p)]\!]^n \geq [\![\Gamma'(p)]\!]^n$ and $[\![\Gamma_t(p)]\!]^N \leq [\![\Gamma'(p)]\!]^N$ for every path p. Hence, $H',\chi',\Gamma' \vdash \Gamma'\checkmark$ holds. Equivalently, $H',\chi',\Gamma' \vdash v : \mathcal{T}(e_b ? e_t : e_f)\checkmark$ follows from the sufficiency $H',\chi',\Gamma_t \vdash : v : \mathcal{T}(e_t)$ in the induction hypothesis and $\mathcal{T}(e_t) \leq e_b ? e_t : e_f$.

Case (S-CONDFALSE): is analogous to (S-CONDTRUE)

Case (S-MEMW∘): Without loss of generality this case assumes $var = \mathtt{x}$.

Since $e_1 = \mathtt{x}$, the rule (S-MEMW) can be simplified to:

$$\frac{\begin{array}{c}\mathtt{x}, H, \chi \xrightarrow[0]{c_{(\text{VARR})}} \underbrace{\chi(\mathtt{x})}_{=\iota}, H, \chi \\ e, H, \chi \xrightarrow[n_2']{n_2} v, H_2, \chi' \\ H' = H_2[\iota \mapsto H_2(\iota)[m \mapsto v]]\end{array}}{\mathtt{x}.m = e, H, \chi \xrightarrow[n_2']{n_2 + c_{(\text{VARR})} + c_{(\text{MEMW})}(m \in \mathtt{Dom}(H_2(\iota)))} v, H', \chi'} \quad (\text{S-MEMW}\circ)$$

Furthermore, since the rule (T-MEMW) requires that $(\mathtt{x} = e')$ is not a subexpression of e it holds that $\chi(\mathtt{x}) = \chi'(\mathtt{x})$.

In this case the induction hypothesis on e states:

$$\mathcal{N}(H, \chi, \Gamma, n) \geq n_2 \tag{6.5}$$

$$\mathcal{N}(H, \chi, \Gamma, n) - \mathcal{N}(H_2, \chi', \Gamma_2, n', v : \mathcal{T}(e)) \geq n_2 - n_2' \tag{6.6}$$

The typing rule states

$$\Gamma_2(\mathtt{x}) = (\mathcal{T}(\mathtt{x}, 1), \bullet) \text{ and}$$
$$\mathcal{T}(\mathtt{x}, 1)(m) = (t_m, \psi_m)$$

The agreement $H_2, \chi', \Gamma_2 \vdash$ follows from the soundness of the underlying system implies

$$H_2, \chi', \Gamma_2 \vdash H_2(\iota) : \mathcal{T}(\mathtt{x}, 1). \tag{6.7}$$

Now assume $\psi_m = \bullet$, then 6.7 implies that $m \in \mathtt{Dom}(H_2(\iota))$ and

$$c_{(\text{MEMW})}(\psi_m = \bullet) = c_{(\text{MEMW})}(true) = c_{(\text{MEMW})}(m \in \mathtt{Dom}(H_2(\iota))) \tag{6.8}$$

In the opposite case $\psi_m = \circ$ the definition (6.3.8) of the constants $c_{(\text{MEMW})}$ implies

$$c_{(\text{MEMW})}(\psi_m = \bullet) = c_{(\text{MEMW})}(false) \geq c_{(\text{MEMW})}(m \in \mathtt{Dom}(H_2(\iota))). \tag{6.9}$$

Combining 6.8 and 6.9 yields in any case

$$c_{(\text{MEMW})}(\psi_m = \bullet) \geq c_{(\text{MEMW})}(m \in \mathtt{Dom}(H_2(\iota))). \tag{6.10}$$

Now the first claimed inequality follows directly:

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n_T)\\
=&\mathcal{N}(H,\chi,\Gamma,n)+c_{(\text{MEMW})}(\psi_m=\bullet)+c_{(\text{VARR})}\\
\overset{6.10}{\geq}&\mathcal{N}(H,\chi,\Gamma,n)+c_{(\text{MEMW})}(m\in \texttt{Dom}(H_2(\iota)))+c_{(\text{VARR})}\\
\overset{6.5}{\geq}&n_2+c_{(\text{MEMW})}(m\in \texttt{Dom}(H_2(\iota)))+c_{(\text{VARR})}\\
=&n_S
\end{aligned}$$

For the second inequality, the preconditions

$$\begin{aligned}
\Gamma_2(var)&=(\mathcal{T}(var,1),\bullet)\\
\Gamma'=\Gamma_2[var&\mapsto(\mathcal{T}(var,2),\bullet)]\\
\mathcal{T}(var,1)&=\mathcal{T}(var,2)
\end{aligned}$$

imply that the contexts Γ_2 and Γ' have the same type paths and for each such path p it holds

$$\Gamma_2(p)=\Gamma'(p) \tag{6.11}$$

Now consider the heap paths $I=\bigcup_{p\in Alias(\iota)} Reach(p)$ in H'. Since H' and H_2 only differ in ι, all other paths $p\in\bar{I}$ in H' also exist in H_2 and it holds

$$\forall p\in\bar{I}:\chi'_{H_2}(p)=\chi'_{H'}(p) \tag{6.12}$$

All paths in H_2 which have been changed in comparison to H' are aliases of paths from v. So consider a path $p=v.m_1.\cdots.m_k$ in H_2. If there is no prefix $p'=v.m_1.\cdots.m_{k'}$ of p with $\chi'_{H_2}(v.m_1.\cdots.m_{k'})=\iota$, then from

$$\chi'_{H_2}(v)=\chi'_{H'}(x.m) \tag{6.13}$$

$$H_2(\iota')=H'(\iota')\forall\iota'\neq\iota \tag{6.14}$$

by the induction on the length of p

$$\begin{aligned}
\chi'_{H_2}(v.m_1.\cdots.m_i,m_{i+1})=&H_2(\chi'_{H_2}(v.m_1.\cdots.m_i))[m_{i+1}]\\
\overset{IH}{=}&H_2(\chi'_{H'}(x.m.m_1.\cdots.m_i))[m_{i+1}]\\
\overset{6.14}{=}&H'(\chi'_{H'}(x.m.m_1.\cdots.m_i))[m_{i+1}]\\
=&\chi'_{H'}(x.m.m_1.\cdots.m_i.m_{i+1})
\end{aligned}$$

it follows

$$\chi'_{H_2}(v.m_1.\cdots.m_k) = \chi'_{H'}(x.m.m_1.\cdots.m_k) \tag{6.15}$$

In the typing context it holds

$$\begin{aligned}
[\![\Gamma_2(v.m_1.\cdots.m_k)]\!]^n =& [\![\mathcal{T}(e)(m_1,...,m_k)]\!]^n \\
=& [\![\mathcal{T}(e)(m_1,...,m_k)]\!]^N \\
=& [\![t_m(m_1,...,m_k)]\!]^N \\
=& [\![t'_m(m_1,...,m_k)]\!]^N \\
=& [\![\mathcal{T}(\mathrm{x},2)(m,m_1,...,m_k)]\!]^N
\end{aligned}$$

Since $\Gamma'(\mathrm{x}) = \mathcal{T}(\mathrm{x},2)$, the sufficiency of Γ' (shown below) implies that this capacity $[\![\mathcal{T}(\mathrm{x},2)(m,m_1,...,m_k)]\!]^N$ is bigger than the sum of the annotations in all alias paths $Alias(x.m.m_1.\cdots.m_k)$. The set $\{p',m,m_1,...,m_k | p' \in Alias(x) = Alias(\mathfrak{l})\}$ is certainly a subset of this alias set:

$$[\![\Gamma_2(v.m_1.\cdots.m_k)]\!]^n = [\![\mathcal{T}(\mathrm{x},2)(m,m_1,...,m_k)]\!]^N \tag{6.16}$$

$$\geq \sum_{p' \in Alias(\mathfrak{l})} [\![\Gamma'(p')(m,m_1,...,m_k)]\!]^n \tag{6.17}$$

If otherwise there is a prefix $v.m_1.\cdots.m_{k'}$ of p with $\chi'_{H_2}(v.m_1.\cdots.m_{k'}) = \mathfrak{l}$, then the preconditions of Lemma 6.4.7 are provided for the address $\mathfrak{l}$ resulting in

$$[\![\Gamma_2(v.m_1.\cdots.m_k)]\!]^n \geq 0 \overset{\text{Lem 6.4.7}}{=} \sum_{p' \in Alias(\mathfrak{l})} [\![\Gamma'(p')(m,m_1,...,m_k)]\!]^n \tag{6.18}$$

From 6.17 and 6.18 this inequality holds true for all paths p of the form $v.m_1.\cdots.m_k$ in H_2. Summing up over all paths $m_1,...,m_k \in Reach(\mathfrak{l})$ with line 6.15 this results in

$$\begin{aligned}
&\sum_{m_1,...,m_k} [\![\Gamma_2(v.m_1.\cdots.m_k)]\!]^n && \geq \sum_{m_1,...,m_k} \sum_{p' \in Alias(\mathfrak{l})} [\![\Gamma'(p')(m,m_1,...,m_k)]\!]^n \\
\Leftrightarrow &\sum_{p \in Reach(v)} [\![\Gamma_2(p)]\!]^n && \geq \sum_{p \in Reach(\mathfrak{l})} \sum_{p' \in Alias(\mathfrak{l})} [\![\Gamma'(p')(m,m_1,....,m_k)]\!]^n \\
\Leftrightarrow &\Sigma_{H_2} v : \mathcal{T}(e) && \geq \sum_{p \in I} [\![\Gamma'(p)]\!]^n
\end{aligned} \tag{6.19}$$

With this, consider the potential

$$\mathcal{N}(H',\chi',\Gamma',n',\mathrm{x}.m : \mathcal{T}(\mathrm{x}.m = e))$$

$$=\Sigma_{H'}\chi' : \Gamma' + n' + \Sigma_{H'}\mathrm{x}.m : \mathcal{T}(\mathrm{x}.m = e)$$

$\mathcal{T}(\mathrm{x}.m = e)^0$ implies $\Sigma_{H'}\mathrm{x}.m : \mathcal{T}(\mathrm{x}.m = e) = 0$

$$\begin{aligned}&=\Sigma_{H'}\chi' : \Gamma' + n'\\ &=\sum_p [\![\Gamma'(p)]\!]^n + n'\end{aligned}$$

Split according to I

$$=\sum_{p\in\bar{I}} [\![\Gamma'(p)]\!]^n + \sum_{p\in I} [\![\Gamma'(p)]\!]^n + n'$$

Γ' is equivalent to Γ_2 regarding annotations

$$\begin{aligned}&\overset{6.19}{\leq} \sum_{p\in\bar{I}} [\![\Gamma_2(p)]\!]^n + \Sigma_{H_2} v : \mathcal{T}(e) + n'\\ &\leq \sum_{p \text{ in } H_2} [\![\Gamma_2(p)]\!]^n + \Sigma_{H_2} v : \mathcal{T}(e) + n'\\ &=\mathcal{N}(H_2, \chi', \Gamma_2, n', v : \mathcal{T}(e))\end{aligned} \tag{6.20}$$

Now the claimed inequality can be derived from the inequality for the subexpressions:

$$\begin{aligned}&\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H',\chi',\Gamma',n'_T, v : \mathcal{T}(\mathrm{x}.m = e))\\ =&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{MEMW})}(\psi_m = \bullet) + c_{(\mathrm{VARR})} - \mathcal{N}(H',\chi',\Gamma',n', v : \mathcal{T}(\mathrm{x}.m = e))\\ \overset{6.20}{\geq}&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\mathrm{MEMW})}(\psi_m = \bullet) + c_{(\mathrm{VARR})} - \mathcal{N}(H_2,\chi',\Gamma_2,n', v : \mathcal{T}(e))\\ \overset{6.6}{\geq}&n_2 + c_{(\mathrm{MEMW})}(\psi_m = \bullet) + c_{(\mathrm{VARR})} - n'_2 = n_S - n'_S\end{aligned}$$

The sufficiency $H',\chi',\Gamma' \vdash v : \mathcal{T}(var.m = e)\checkmark$ is equivalent to $H',\chi',\Gamma' \vdash v : t_m\checkmark$ due to $\mathcal{T}(var.m = e)\ {}_N{=}_N\ \mathcal{T}(e)\ {}_N{=}_N\ t_m$. Since $\Gamma'(x.m) = t_m$ and $\chi'_{H'}(x.m) = v$ this is contained in $H',\chi',\Gamma' \vdash \checkmark$.

It remains to show the sufficiency $H',\chi',\Gamma' \vdash \checkmark$. Since H' and H_2 only differ in $\iota.m$, all paths which do not have an alias path passing though $\iota.m$ are sufficient by the induction hypothesis on e. Furthermore, assume a path p has an alias passing through $\iota.m$ but does not pass through $\iota.m$ itself. In this case, in H_2 the path p has a non-trivial prefix being an alias to v and forms a loop in the heap. From Lemma 6.4.7 it follows that $[\![\Gamma_2(p)]\!]^n = 0$. The same argument prohibits annotated loops within v: if a pair of

paths $v.p, v.p'$ are aliases, i.e. $\chi'_{H_2}(v.p) = \chi'_{H_2}(v.p')$, then

$$\begin{aligned}[\![\Gamma_2(v.p)]\!]^n =& [\![\Gamma_2(v.p)]\!]^N \\ \geq& \sum_{p' \in Alias(v.p)} [\![\Gamma_2(v.p')]\!]^n \\ \geq& [\![\Gamma_2(v.p)]\!]^n + [\![\Gamma_2(v.p')]\!]^n\end{aligned}$$

This implies $[\![\Gamma_2(v.p)]\!]^n = 0$ and $[\![\Gamma_2(v.p')]\!]^n = 0$.

Now assume a path p with an alias through $\iota.m$ in H':

$$\begin{aligned}\sum_{p' \in Alias_{H'}(p)} [\![\Gamma'(p')]\!]^n =& \sum_{p' \in Alias_{H'}(p)} [\![\Gamma_2(p')]\!]^n \\ =& \sum_{\substack{p' \in Alias_{H'}(p) \\ p' \text{ through } \iota.m}} [\![\Gamma_2(p')]\!]^n\end{aligned}$$

Every such path p' can be written as $p_1 \cdot p_2$ with $\chi'_{H'}(p_1) = H'(\iota).m$ and p_1 is the shortest such path (i.e. no prefix of p_1 also visits $H'(\iota).m$). The remaining path p_2 is a path in v. All positively annotated paths in v have no aliases. Therefore, all important paths in this sum must have the same path p_2.

$$= \sum_{p_1 \in Alias_{H'}(\iota.m)} [\![\Gamma_2(p_1)(p_2)]\!]^n$$

All these paths $p' \in Alias(\iota.m)$ also exist equally in H_2

$$= \sum_{p_1 \in Alias_{H_2}(\iota.m)} [\![\Gamma_2(p_1)(p_2)]\!]^n$$

Choose one such p_1. Since p_1 is sufficient:

$$\begin{aligned}\leq& [\![\Gamma_2(p_1)(p_2)]\!]^N \\ \leq& [\![\Gamma'(p_1)(p_2)]\!]^N \\ \leq& [\![\Gamma'(p_1)]\!]^N \\ =& [\![\Gamma'(p)]\!]^N\end{aligned}$$

The last equality holds, since in the rules aliases are only created by the sharing relation, which requires each alias to have the same capacities. The capacities are not changed later on.

This is exactly the definition of "p is sufficient" for all p that were not already sufficient from H_2.

Case S-memW•: The same argumentation as in the case S-SEQ leads to bounds:

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n) && \geq n_1\\
&\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H_1,\chi_1,\Gamma_1,n',\iota:\mathcal{T}(e_1)) && \geq n_1 - n_1'\\
&\mathcal{N}(H_1,\chi_1,\Gamma_1,n') && \geq n_2\\
&\mathcal{N}(H_1,\chi_1,\Gamma_1,n') - \mathcal{N}(H_2,\chi',\Gamma',n'',v:\mathcal{T}(e_2)) && \geq n_2 - n_2'\\
&\mathcal{N}(H,\chi,\Gamma,n) && \geq n_\wp\\
&\mathcal{N}(H,\chi,\Gamma,n) - \mathcal{N}(H_2,\chi',\Gamma',n'',v:\mathcal{T}(e_2)) && \geq n_\wp - n_\wp' \qquad (6.21)
\end{aligned}$$

The same argumentation as in the case S-MEMW∘ applies and directly implies the first claimed inequality

$$\mathcal{N}(H,\chi,\Gamma,n_T) \geq n_S$$

The equivalent split of the set of heap paths into $I, \bar{I}$ and I' results in

$$\begin{aligned}
&\mathcal{N}(H',\chi',\Gamma',n',v:\mathcal{T}(e_1.m = e_2))\\
\leq&\mathcal{N}(H_2,\chi',\Gamma',n',v:\mathcal{T}(e_2)) \qquad (6.22)
\end{aligned}$$

which results in the second claimed inequality:

$$\begin{aligned}
&\mathcal{N}(H,\chi,\Gamma,n_T) - \mathcal{N}(H',\chi',\Gamma',n_T',v:\mathcal{T}(e_1.m = e_2))\\
=&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\text{MEMW})}(true) - \mathcal{N}(H',\chi',\Gamma',n',v:\mathcal{T}(e_1.m = e_2))\\
\overset{6.22}{\geq}&\mathcal{N}(H,\chi,\Gamma,n) + c_{(\text{MEMW})}(true) - \mathcal{N}(H_2,\chi',\Gamma',n',v:\mathcal{T}(e_2))\\
\overset{6.21}{\geq}&n_\wp + c_{(\text{MEMW})}(m \in \texttt{Dom}(H_2(\iota))) - n_\wp' = n_S - n_S'
\end{aligned}$$

With the same preconditions as in the case S-MEMW∘ the sufficiency is equivalent.

□

6.5 Type inference

The type inference algorithm infers the amortised types by performing the following steps.

1. The data types are computed by the underlying system. Depending on the underlying system this step might require additional code annotations.

2. The data types are promoted to amortised types by recursively injecting annotation variables. Where the typing rules use multiple copies of a data type $\mathcal{T}(e,1), \mathcal{T}(e,2)...$ this data type needs to be copied.

3. The typing rules of AmorJiSe are applied and the resulting constraints on the resource annotation variables are collected into a set of linear constraints. This also includes the implicit constraints $n \geq 0$ for every annotation variable n. Since the structures of the data types are known, all higher level constraints $(t \,{}_n\!\leq_N t, t \hookrightarrow t \oplus t, ...)$ can be translated into linear constraints.

4. The resulting set of constraints is solved as a Linear Programming Problem (LPP) with conventional solvers.

5. The bounds on the resource consumption of the whole program are extracted from the inferred annotation values of the initial context and the type of the functions.

The type lookup $\mathcal{T}(\cdot)$ used in inference, in addition to inserting annotation variables into the data types of the underlying system, also ensures that all contained types are unrolled on the highest level. For example, the list data type $\mu\alpha.[\texttt{value} : (\texttt{Int}, \bullet), \texttt{next} : (\alpha, \bullet)]$ is expanded into

$$\mu\beta.[\texttt{value} : ((\texttt{Int}, n_1/N_1), \bullet), \texttt{next} : ((\mu\alpha.[\texttt{value} : ((\texttt{Int}, n_3/N_3), \bullet), \texttt{next} : ((\alpha, n_4/N_4), \bullet)] , n_2/N_2), \bullet)]$$

Type unrolling does not change the soundness of a type. The different annotation variables for the head $n_1/N_1, n_2/N_2$ and the recursive end $n_3/N_3, n_4, N_4$ enable an expression to use the reserved resource units in the head without effecting all fields in the recursive structure. This way more programs are typeable.

The inference procedure is comparable to the type inference presented in previous work on amortised type systems [58] by Hofmann et al. The method proposed in this work infers the amortised annotations for a class-based object-oriented programming language similar to Java called RAJA. The big difference between the RAJA inference and AmorJiSe is the object type representation. While AmorJiSe uses one individual object expression for each object value, RAJA types multiple expressions with the same class. To differentiate between different resource properties of values types with

the same class, Hofmann et al. have to introduce views which assign a specific set of annotations to a value typed with a class.

Comparing the procedure, the RAJA inference can skip the first step, since the data types are supplied as part of language's syntax. The second step is solved implicitly, by defining the $\diamond$ operator. It returns the appropriate resource annotations for each class-view combination and its fields. Since the annotation variables are represented implicitly using this operator, they are not injected into the types. RAJA then generates constraints equivalent to the third step of AmorJiSe, including the higher level constraints (see Section 6.3.6.6) on infinite trees. They are, however, complicated due to the view notation. These constraints are solved similarly converted into arithmetic inequalities and solved by conventional LPP solving methods.

6.5.1 Annotation domain

Each annotation is interpreted as the amount of reserved resource units stored with a typed value. Intuitively, the annotations are therefore thought to be integers. However, Integer Linear Programming Problems (ILPP) are NP-hard, whereas a real-valued solution to the existing Linear Programming Problem can be computed in polynomial time. To profit from the better complexity, AmorJiSe allows annotation values in $\mathbb{R}$.

In the resource model this has the following interpretation. A resource unit can be split into arbitrary pieces and each piece stored in different data structures during execution. To access the resource a full unit is recombined from the pieces available in the data structures at the point of access and used to pay for the access. Even though the unit is potentially split and transported via different data structures only full units are used to legitimate resource access. Since the soundness only states that the sum of resource units stored in the values is bigger than the number of resource accesses, this method is sound independent of the splitting of the units. Also note that the solution found with the LPP algorithm is always smaller than the solution found by an ILPP algorithm, since the minimal ILPP solution also is a solution for the LPP algorithm. On the other hand smaller LPP solutions might exists which contain non-integers.

A second consideration is the inclusion of ∞ as annotation. If one of the annotations in the data-structures in the initial typing context needs to be infinite in the minimal solution for the LPP, then the resulting overall resource bound is infinite as well. This situation is equivalent to the solver not being able to compute a finite bound. Therefore, allowing infinity as value for the annotations contained in the value types does not add

expressivity.

On the other hand, it is possible to allow infinity as annotation in a function type. If an input handler function (assuming it is not otherwise called) is typed with an infinite annotations as requirement, the overall bound is finite, but the interaction-dependent part of the bound is infinite as soon as the associated input event occurred at least once. Therefore, allowing annotations inside the function types to be ∞ results in certain situations in an improvement of the precision of the bound: the bound expresses which user interactions make the consumption potentially infinite and provides a finite bound for the cast these interactions do not occur. This, however, depends on the ability of the used LPP solver to derive ∞ as value for the variables, which are not contained in the objective function.

6.5.2 Deriving bounds

Given the result $\Gamma, n \vdash e : t | \Gamma', n'$ the bound on the resource consumption of the expression e consists of 3 parts:

1. The annotations n, n' describe the *constant* resource consumption.

2. The potential $\sum_H \chi : \Gamma$ describes the *data-dependent* resource consumption, if e is executed with the inputs stored in H, χ.

3. The function types in Γ' describe the *interaction-dependent* resource consumption: If for example e registers a function f as handler for the event `click` and $\Gamma'(f) = (((O \times t, n \to t', n'), n_f / N_f), \psi_f)$ then each `click` event has a resource consumption of (n, n').

6.6 Extensions

The core language and type system discussed so far is not very expressive. This section increases the flexibility of the annotations and adds rules for functions.

6.6.1 Exchange rule

The system shown above tracks the resource consumption of a typed expression via the annotations n, n' of the typing judgement. Additionally it uses the annotations injected into the types to express resource units dependent on the size of data structures.

However, in the rules presented above the two different kinds of annotations are separate. The type judgement annotations "pay" for resource accesses but the amortised annotations cannot be transferred into the type judgement. This separation simplifies the proof of Theorem 6.4.10. The following rule allows to swap resource units from the amortised annotations into the typing judgement annotations and therefore makes the reserved resource units usable.

$$\frac{\Gamma, n \vdash e : (t, n_t/N_t) | \Gamma', n' \qquad 0 \leq r \leq n_t}{\Gamma, n \vdash e : (t, n_t - r/N_t) | \Gamma', n' + r} \tag{T-SWAP}$$

Theorem 6.6.1. *The rule (*T-SWAP*) preserves Theorem 6.4.10*

Proof. The proof extends the induction step of 6.4.10

Case (T-SWAP): For the potential it holds:

$$\mathcal{N}(H', \chi', \Gamma', n', v : (t, n_t/N_t)) = \mathcal{N}(H', \chi', \Gamma', n' + r, v : (t, n_t - r/N_t))$$

The required inequalities follow directly from the induction hypothesis. For the sufficiency of Γ', consider a path p in the state $H, \chi, \Gamma, v : (t, n_t/N_t)$. Due to the induction hypothesis it is sufficient, which means $[\![\Gamma_{(t,n_t/N_t)}(p)]\!]^N \geq \sum_{p' \in Alias(p)} [\![\Gamma_{(t,n_t/N_t)}(p')]\!]^n$. The only difference between $\Gamma_{(t,n_t/N_t)}(\cdot)$ and $\Gamma_{(t,n_t-r/N_t)}(\cdot)$ is the annotation of the path v. If $v \notin Alias(p)$, then

$$\begin{aligned}
&[\![\Gamma_{(t,n_t-r/N_t)}(p)]\!]^N \\
=&[\![\Gamma_{(t,n_t/N_t)}(p)]\!]^N \\
\overset{IH}{\geq}& \sum_{p' \in Alias(p)} [\![\Gamma_{(t,n_t/N_t)}(p')]\!]^n \\
=& \sum_{p' \in Alias(p)} [\![\Gamma_{(t,n_t-r/N_t)}(p')]\!]^n
\end{aligned}$$

Otherwise, if $v \in Alias(p)$

$$\begin{aligned}
&[\![\Gamma_{(t,n_t-r/N_t)}(p)]\!]^N \\
=&[\![\Gamma_{(t,n_t/N_t)}(p)]\!]^N \\
\overset{IH}{\geq}& \sum_{p' \in Alias(p)} [\![\Gamma_{(t,n_t/N_t)}(p')]\!]^n \\
=& \sum_{p' \in Alias(p)} [\![\Gamma_{(t,n_t-r/N_t)}(p')]\!]^n + r \\
\overset{r>0}{\geq}& \sum_{p' \in Alias(p)} [\![\Gamma_{(t,n_t-r/N_t)}(p')]\!]^n
\end{aligned}$$

Since this is true for arbitrary p, this constitutes to $H', \chi', \Gamma' \vdash \Gamma' \checkmark$.

The sufficiency of v in $H, \chi, \Gamma, v : (t, n_t/N_t)$ is provided by the induction hypothesis and provides the sufficiency of any path $p \in Reach(v)$ in $H, \chi, \Gamma, v : (t, n_t - r/N_t)$ as above except for the path v:

$$
\begin{aligned}
& [\![\Gamma_{(t,n_t-r/N_t)}(v)]\!]^N \\
=& [\![\Gamma_{(t,n_t/N_t)}(v)]\!]^N \\
\geq & \sum_{p' \in Alias(v)} [\![\Gamma_{(t,n_t/N_t)}(p')]\!]^n - r \\
\geq & \sum_{p' \in Alias(v)-\{v\}} [\![\Gamma_{(t,n_t/N_t)}(p')]\!]^n + [\![\Gamma_{(t,n_t/N_t)}(v)]\!]^n - r \\
\geq & \sum_{p' \in Alias(v)-\{v\}} [\![\Gamma_{(t,n_t-r/N_t-r)}(p')]\!]^n + [\![\Gamma_{(t,n_t-r/N_t-r)}(v)]\!]^n \\
= & \sum_{p' \in Alias(v)} [\![\Gamma_{(t,n_t-r/N_t-r)}(p')]\!]^n
\end{aligned}
$$

□

The difference between the rule (T-SWAP) and the other rules of AmorJiSe is that (T-SWAP) is not syntax directed, but makes the rule application non-deterministic in two ways: The derivation has to guess when the rule (T-SWAP) is applied and the value of the parameter r for each application. This non-determinacy makes type checking without provided type derivation more difficult. The complexity of type inference, however, does not increase significantly. Applying the new rule twice can be collapsed into one application by adding the r values of the two applications. Therefore, type inference applies the rule (T-SWAP) exactly once after the application of every other rule. This results in one extra variable (the parameter r) and two additional linear constraints for each applied syntax-driven rule. In the worst case this doubles the size of the LPP.

The same result could be achieved by incorporating the rule (T-SWAP) into each of the rules provided above. However, this would blow up the representation of the rules.

6.6.2 Functions

As functions do not interfere with the object structures of the types in AmorJiSe, they have been omitted from the core system described above and are added in the following. This extension handles non-nested, potentially recursive, first order functions.

Method calls can be handled similarly as shown in Section 6.7.1. The syntax of a program with functions consists of a list of function definitions and a main expression e as before.

$$P ::= F^*; e \qquad \text{(Program)}$$

$$F ::= \texttt{function} f(\texttt{x})\{e\} \qquad \text{(FuncDecl)}$$

Function values are noted as $v = \texttt{function} f(x) e_f$, where f is the function name, x the function argument and e_f the function body expression. Similar to the restriction to one variable x in the JavaScript syntax above, function definition here are restricted to one argument named x for simplicity reasons.

In JavaScript function statements are interpreted in a first pass: before an expression is evaluated, its source code is scanned for functions and the function definition for each function is added to the scope. This makes it possible to define mutually recursive functions. AmorJiSe assumes that the function declarations precede the main expression e. This automatically enforces this first pass behaviour. A program not in this shape can be translated without change in behaviour by reordering the source code. Furthermore, AmorJiSe restricts functions to non-nested function declarations. This ensures that all functions are executed in the same scope and simplifies the presentation and the soundness proof significantly. Programs with nested function declarations can be flattened by encoding the scope of each local variable into the variable name.

6.6.2.1 Typing rules

Figure 6.6 and 6.7 introduce the rules to handle functions in AmorJiSe: the rules (S-PROGRAM) / (T-PROGRAM) handle function statements and the rules (S-FUNX) / (T-FUNX) handle function calls.

During evaluation, the definition of the functions is stored in the scope χ, such that the rule (T-FUNX) can lookup the function body. The typing rules equivalently store the type for function f as $\Gamma(f)$. Although this adds to $\texttt{Dom}(\chi)$ and $\texttt{Dom}(\Gamma)$ this does not invalidate the proof of Theorem 6.4.10. Since the added values are only functions, they only add trivial paths and since they are always typed with the amortised annotation $0/0$, they can be ignored for the computation of the potential.

During the analysis of a function definition, the bodies of the functions are analysed in the context Γ_i, which only contains the types of the functions (from Γ_F) and the

Figure 6.6 Evaluation rules - Functions

$$\frac{\begin{array}{c} F_i = \texttt{function} f_i(\texttt{x})\{e_i\} \\ \chi_F = \chi[f_i \mapsto F_i \quad \forall i = 1..k] \\ e, H, \chi_F \xrightarrow[n'_e]{n_e} v, H', \chi' \end{array}}{F_1, ..., F_k; e, H, \chi \xrightarrow[n'_e]{n_e + k \cdot c_{(\text{FUND})}} v, H', \chi'} \qquad \text{(S-PROGRAM)}$$

$$\frac{\begin{array}{c} e, H, \chi \xrightarrow[n'_1]{n_1} v_e, H_1, \chi_1 \\ \chi(f) = \texttt{function}(\texttt{x})\{e_f\} \\ \chi_f = \{f_i \mapsto \chi_1(f_i) \forall \text{functions } f_i, \texttt{x} \mapsto v_e, \texttt{this} \mapsto \{\}\} \\ e_f, H_1, \chi_f \xrightarrow[n'_2]{n_2} v, H', \chi' \\ (n_\S, n'_\S) = (n_1, n'_1) \S (n_2, n'_2) \end{array}}{f(e), H, \chi \xrightarrow[n'_\S]{n_\S + c_{(\text{FUNX})}} v, H', \chi_1} \qquad \text{(S-FUNX)}$$

parameters for this specific function. Therefore, the typing context, the function body is typed in, does not depend on the calling context, which simplifies the soundness proof for the function call.

The existing type relations extend for function types in the intuitive way. The constraints $G_1 \,\eta \leq_{\eta'} G_2$ produce for function types $G = O \times t_x, n \to t_{ret}, n'$ the linear constraints $n_1 = n_2$ and $n'_1 = n'_2$ as well as all constraints generated by $t_{x1} \,\eta =_{\eta'} t_{x2}$, $O \,\eta =_{\eta'} O$ and $t_{ret1} \,\eta =_{\eta'} t_{ret2}$. The sharing relation $G_1 \hookrightarrow G_2 \oplus G_3$ also produces equality constraints for all three involved types and the potential is defined as $\Sigma_H v : G = 0$.

6.6.2.2 Soundness

For the soundness statement, first the agreement relation has to be extended.

Definition 6.6.2. Given a state H, χ, Γ, a value v agrees with its type t written as $H, \chi, \Gamma \vdash v : t$, if - in addition to the requirements in Definition 6.4.8 - for every path p with $t(p) = O \times t_{\text{x}}, n_f \to t_{ret}, n'_f$ provided

- $\Gamma_f = [\texttt{x} \mapsto t_{\text{x}}, \texttt{this} \mapsto O]$
- any χ, H with $H, \chi, \Gamma \vdash$
- any $v_{ret}, H', \chi', n_S, n'_S$ with $H, \chi, e \xrightarrow[n'_S]{n_S} H', \chi', v_{ret}$

it holds that

Figure 6.7 Typing rules - Functions

$$\frac{\begin{array}{c} F_i = \texttt{function} f_i(\texttt{x}) e_i \\ \Gamma_F = \Gamma[f_i \mapsto (\mathcal{T}(f_i), \bullet) \forall i = 1..k] \\ \mathcal{T}(f_i) = ((O_i \times t_i, n_i \to t_i', n_i'), 0/0) \\ \Gamma_i = \Gamma_F[\texttt{x} \mapsto t_i, \texttt{this} \mapsto O_i] \\ \Gamma_i, n_i \vdash e_i : \mathcal{T}(e_i) | \Gamma_i', n_i' \\ \Gamma_1, n \vdash e : \mathcal{T}(e) | \Gamma', n' \end{array}}{\Gamma, n + k \cdot c_{(\text{FUND})} \vdash F_1; ...; F_k; e : \mathcal{T}(e) | \Gamma', n'} \quad \text{(T-PROGRAM)}$$

$$\frac{\begin{array}{c} \Gamma, n \vdash e : \mathcal{T}(e) | \Gamma_1, n_e \\ \Gamma_1(f) = O \times t_{\text{x}}, n_f \to t_{ret}, n_f' \\ O \text{ does not contain } \bullet \text{ fields} \\ O^0 \qquad \mathcal{T}(e) \leq t_{\text{x}} \qquad \mathcal{T}(e) \,_N{=}_N\, t_{\text{x}} \\ t_{ret} \leq \mathcal{T}(f(e)) \\ n_e \geq n_f \qquad n' = n_e - n_f + n_f' \end{array}}{\Gamma, n + c_{(\text{FUNX})} \vdash f(e) : \mathcal{T}(f(e)) | \Gamma_1, n'} \quad \text{(T-FUNX)}$$

- $H', \chi', \Gamma \vdash v_{ret} : t_{ret}$
- $n_f \geq n_S$
- $n_f - n_f' \geq n_S - n_S'$

Theorem 6.6.3. *The rules* (T-PROGRAM), (T-FUNX), (S-PROGRAM) *and* (S-FUNX) *preserve Theorem 6.4.10 with the additional invariant:*

For every function f with

$$\chi(f) = \texttt{function} f(x)\{e\}$$
$$\Gamma(f) = O \times t_x, n_f \to t_{ret}, n_f'$$

the body e can be typed as:

$$\Gamma_f, n_f \vdash e : t_{ret} | \Gamma', n_f'$$

with $\Gamma_f = \{f_i \mapsto \Gamma(f_i), x \mapsto t_x, \texttt{this} \mapsto O\}$

Proof. This proof extends the induction step of Theorem 6.4.10. The assumption (Theorem 6.4.9) applies with the amended agreement relation. Since the rules of the original proof do not effect the functions stored in Γ or χ, they trivially fulfil the new invariant.

Case (S-PROGRAM): Due to the preconditions of the soundness theorem it is given that Γ is sufficient for H, χ. This relation obviously extends to Γ_1 and H, χ_1, since each G_i is sufficient with the annotation $0/0$ and no aliases nor fields. The potential inequality can be proven using the induction hypothesis:

$$\begin{aligned}
&\mathcal{N}(H, \chi, \Gamma, n + k \cdot c_{(\text{FUND})}) \\
=&\mathcal{N}(H, \chi_1, \Gamma_1, n) + k \cdot c_{(\text{FUND})} \\
\geq& n_S + k \cdot c_{(\text{FUND})} \\
&\mathcal{N}(H, \chi, \Gamma, n + k \cdot c_{(\text{FUND})}) - \mathcal{N}(H', \chi', \Gamma', v : t_e, n') \\
=&\mathcal{N}(H, \chi_1, \Gamma_1, n) - \mathcal{N}(H', \chi', \Gamma', v : t_e, n') + k \cdot c_{(\text{FUND})} \\
\geq& n_S - n'_S + k \cdot c_{(\text{FUND})}
\end{aligned}$$

The sufficiency of Γ' for H', χ' follows directly from the induction hypothesis.

Furthermore, note that rule (T-PROGRAM) explicitly requires the function invariant to be true for each function in the resulting environment H', χ', Γ'.

Case (S-FUNX):

Due to the induction hypothesis on e the following holds:

$$\mathcal{N}(H, \chi, \Gamma, n) \geq n_1 \tag{6.23}$$

$$\mathcal{N}(H, \chi, \Gamma, n) + \mathcal{N}(H_1, \chi_1, \Gamma_1, n_e, v_e : \mathcal{T}(e)) \geq n_1 + n'_1 \tag{6.24}$$

From the soundness of the underlying system it follows that v_e agrees with $\mathcal{T}(e)$. Since O^0 and $\mathcal{T}(e) \leq t_{\mathtt{x}}$ it is clear that the values $\chi_f(\mathtt{x}) = v_e$ and $\chi_f(this) = \{\}$ agree with the types O and $t_{\mathtt{x}}$. Construct the context $\Gamma_f = \{f_i \mapsto \Gamma(f), \mathtt{x} \mapsto t_{\mathtt{x}}, this \mapsto O\}$ which fulfils the agreement relation $H_1, \chi_f, \Gamma_f \vdash$. From the function invariant it follows that $\Gamma_f, n_f \vdash e_f : t_{ret} | \Gamma'_f, n'_f$ and by the induction hypothesis it follows

$$\mathcal{N}(H_1, \chi_f, \Gamma_f, n_f) \geq n_2 \tag{6.25}$$

To prove the needed properties about the types $\Gamma_1(\mathtt{x})$ and $\Gamma_1(\mathtt{this})$ in the resulting heap H' they are carried through the execution of the function body. For this reason, with fresh variables $\mathtt{this}_0$ and $\mathtt{x}_0$ construct the following extended states:

$$\begin{aligned}
\Gamma_+ &= \Gamma_f[\mathtt{x}_o \mapsto \Gamma_1(\mathtt{x}), \mathtt{this}_o \mapsto \Gamma_1(\mathtt{this})] \\
\Gamma'_+ &= \Gamma'_f[\mathtt{x}_o \mapsto \Gamma_1(\mathtt{x}), \mathtt{this}_o \mapsto \Gamma_1(\mathtt{this})] \\
\chi_+ &= \chi_f[\mathtt{x}_o \mapsto \chi_1(\mathtt{x}), \mathtt{this}_o \mapsto \chi_1(\mathtt{this})] \\
\chi'_+ &= \chi'[\mathtt{x}_o \mapsto \chi_1(\mathtt{x}), \mathtt{this}_o \mapsto \chi_1(\mathtt{this})]
\end{aligned}$$

which fulfil the agreement relation $H_1, \chi_+, \Gamma_+ \vdash$ due to $H_1, \chi_f, \Gamma_f \vdash$ and $H_1, \chi_1, \Gamma_1 \vdash$. Since the expression e_f does not access the variables $\mathtt{x}_o, \mathtt{this}_o$ it is clear that

$$\Gamma_+, n_f \vdash e_f : t_{ret} | \Gamma'_+, n'_f$$
$$e_f, H_1, \chi_+ \xrightarrow[n'_2]{n_2} \!\!\!\!\rightarrow v, H', \chi'_+$$

are equivalent to the evaluation in Γ_f, χ_f. With these typing relations and the evaluation of e_f the induction hypothesis results in

$$\mathcal{N}(H_1, \chi_+, \Gamma_+, n_f) \geq n_2 \tag{6.26}$$
$$\mathcal{N}(H_1, \chi_+, \Gamma_+, n_f) - \mathcal{N}(H', \chi'_+, \Gamma'_+, n'_f, v : t_{ret}) \geq n_2 - n'_2 \tag{6.27}$$

By construction, the relationship between the original state and the extended state is the following:

$$\Sigma_{H_1}\chi_+ : \Gamma_+ = \Sigma_{H_1}\chi_f : \Gamma_f + \Sigma_{H_1}\chi_1 : \Gamma_1 \tag{6.28}$$
$$\Sigma_{H'}\chi'_+ : \Gamma'_+ = \Sigma_{H'}\chi' : \Gamma'_f + \Sigma_{H'}\chi_1 : \Gamma_1 \tag{6.29}$$

Since $\chi_f(\mathtt{this}) = \{\}$ it is clear that $\Sigma_{H_1}\chi_f(\mathtt{this}) : \Gamma_f(\mathtt{this}) = 0$ and therefore $\Sigma_{H_1}\chi_f : \Gamma_f = \Sigma_{H_1} v_e : t_{\mathtt{x}} \overset{\mathcal{T}(e) \leq t_{\mathtt{x}}}{\leq} \Sigma_{H_1} v_e : \mathcal{T}(e)$. So

$$\Sigma_{H_1}\chi_f : \Gamma_f - \Sigma_{H_1} v_e : \mathcal{T}(e) \leq 0 \tag{6.30}$$

From this and the constraint

$$n_e \geq n_f \tag{6.31}$$

in the typing rule it follows:

$$\mathcal{N}(H_1, \chi_1, \Gamma_1, n_e, v_e : \mathcal{T}(e)) \tag{6.32}$$
$$= \Sigma_{H_1}\chi_1 : \Gamma 1 + n_e + \Sigma_{H_1} v_e : \mathcal{T}(e) \tag{6.33}$$
$$\overset{6.31,6.30}{\geq} \Sigma_{H_1}\chi_1 : \Gamma_1 + n_f + \Sigma_{H_1}\chi_f : \Gamma_f \tag{6.34}$$
$$\geq n_f + \Sigma_{H_1}\chi_f : \Gamma_f = \mathcal{N}(H_1, \chi_f, \Gamma_f, n_f) \tag{6.35}$$

The typing rule has the constraint

$$n' = n_e - n_f + n'_f \tag{6.36}$$

Now the inequalities can be shown. The first inequality is shown in cases again: If $n_1' \geq n_2$ then $n_\wr = n_1$ and it holds

$$\begin{aligned}&\mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})})\\ \overset{6.23}{\geq}\ & n_1 + c_{(\text{FUNX})}\\ =\ & n_\wr + c_{(\text{FUNX})} = n_S\end{aligned}$$

In the other case $n_1' \leq n_2$ it holds $n_\wr = n_1 - n_1' + n_2$:

$$\begin{aligned}&\mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})})\\ \overset{6.32}{\geq}\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \mathcal{N}(H_1,\chi_1,\Gamma_1,n_e,v_e:\mathcal{T}(e)) + \mathcal{N}(H_1,\chi_f,\Gamma_f,n_f)\\ \overset{6.25,6.24}{\geq}\ & n_1 - n_1' + n_2 + c_{(\text{FUNX})}\\ =\ & n_\wr + c_{(\text{FUNX})} = n_S\end{aligned}$$

The second inequality can be proven as follows:

$$\begin{aligned}&\mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \mathcal{N}(H',\chi_1,\Gamma_1,n',v:t_{ret})\\ =\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \Sigma_{H'}\chi_1:\Gamma_1 - n' - \Sigma_{H'}v:t_{ret}\\ \geq\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \Sigma_{H'}\chi_1:\Gamma_1 - n' - \Sigma_{H'}v:t_{ret} - \Sigma_{H'}\chi':\Gamma_f'\\ \overset{6.36}{=}\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \Sigma_{H'}\chi_1:\Gamma_1 - n_e + n_f - n_f' - \Sigma_{H'}v:t_{ret} - \Sigma_{H'}\chi':\Gamma_f'\\ \overset{6.30}{\geq}\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \Sigma_{H'}\chi_1:\Gamma_1 - n_e + n_f - n_f' - \Sigma_{H'}v:t_{ret} - \Sigma_{H'}\chi':\Gamma_f'\\ & \quad - \Sigma_{H_1}v_e:\mathcal{T}(e) + \Sigma_{H_1}\chi_f:\Gamma_f\\ =\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \Sigma_{H'}\chi_1:\Gamma_1 - n_e + n_f - n_f' - \Sigma_{H'}v:t_{ret} - \Sigma_{H'}\chi':\Gamma_f'\\ & \quad - \Sigma_{H_1}v_e:\mathcal{T}(e) + \Sigma_{H_1}\chi_f:\Gamma_f - \Sigma_{H_1}\chi_1:\Gamma_1 + \Sigma_{H_1}\chi_1:\Gamma_1\\ \overset{6.28,6.29}{=}\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \Sigma_{H_1}\chi_1:\Gamma_1 - n_e + n_f - n_f' - \Sigma_{H'}v:t_{ret} - \Sigma_{H'}\chi_+':\Gamma_+'\\ & \quad - \Sigma_{H_1}v_e:\mathcal{T}(e) + \Sigma_{H_1}\chi_+:\Gamma_+\\ =\ & \mathcal{N}(H,\chi,\Gamma,n+c_{(\text{FUNX})}) - \mathcal{N}(H_1,\chi_1,\Gamma_1,n_e,v_e:\mathcal{T}(e))\\ & \quad + \mathcal{N}(H_1,\chi_+,\Gamma_+,n_f) - \mathcal{N}(H',\chi_+',\Gamma_+',n_f',v:t_{ret})\\ \overset{6.24,6.27}{\geq}\ & n_1 + c_{(\text{FUNX})} - n_1' + n_2 - n_2' = n_\wr + c_{(\text{FUNX})} - n_\wr'\end{aligned}$$

The claimed inequalities follow directly from $t_x \leq \mathcal{T}(f(e))$ and the implied $\Sigma_{H'}v : \mathcal{T}(f(e)) \leq \Sigma_{H'}v : t_x$.

Now it remains to show that the final state is sufficient: The induction hypothesis applies to e and implies that Γ_1 is sufficient for H_1. Now, the sufficiency has to be extended to the state the function body is executed in. Since the functions $f_1, \ldots, f_k$

of the analysed program cannot be overwritten, we know that the obtained type $O \times t_{\mathtt{x}} n_f \to t_{ret}, n'_f$ still fits the function body e. Under this circumstance, the function invariant established that in the context $\Gamma_f = \Gamma_P[\mathtt{x} \mapsto t_{\mathtt{x}}, \mathtt{this} \mapsto O]$ the function body e_f can be typed as t_{ret}. This context Γ_f is sufficient for the state H_1, χ_f: this only has to be shown for the paths $\mathtt{x}, \mathtt{this}$. The sufficiency for the remaining paths then follows from Lemma 6.4.4. For the path $\mathtt{x}$ the value v_e and type $t_{\mathtt{x}}$ is known and $t_{\mathtt{x}}$ is sufficient since $\mathcal{T}(e)$ is sufficient due to the induction hypothesis on e and $\mathcal{T}(e) \leq t_{\mathtt{x}}$. From the requirement "$O$ contains no $\bullet$ fields", it follows that the object type O agrees with the empty object value $\{\}$. The only valid heap path for $\{\}$ is $\mathtt{this}$. Due to O^0, this type O is trivially sufficient for this value. O might still contain additional $\circ$ fields, which are assigned to in the function body.

Altogether this shows that Γ_f is sufficient for χ_f, H_1. The induction hypothesis is applicable on e_f and the state H_1, χ_f, Γ_f. Now choose new variables $\mathtt{x}_{outer}$ and $\mathtt{this}_{outer}$, which are not accessed in e_f and the extended state $(H_1, \bar{\chi}_f, \bar{\Gamma}_f)$ defined with

$$\bar{\chi}_f = \chi_f[\mathtt{x}_{outer} \mapsto \chi_1(\mathtt{x}), \mathtt{this}_{outer} \mapsto \chi_1(\mathtt{this})] \text{ and}$$
$$\bar{\Gamma}_f = \Gamma_f[\mathtt{x}_{outer} \mapsto \Gamma_1(\mathtt{x}), \mathtt{this}_{outer} \mapsto \Gamma_1(\mathtt{this})]$$

Since e_f does not access $\mathtt{this}_{outer}$ nor $\mathtt{x}_{outer}$ the results from the preconditions can be extended to

$$e_f, H_1, \bar{\chi}_f \xrightarrow[n'_2]{n_2} v, H', \bar{\chi}'$$
$$\bar{\Gamma}_f, n_f \vdash e_f : t_{ret} | \bar{\Gamma}'_f, n'_f$$

where

$$\bar{\chi}' = \chi'[\mathtt{x}_{outer} \mapsto \chi_1(\mathtt{x}), \mathtt{this}_{outer} \mapsto \chi_1(\mathtt{this})] \text{ and}$$
$$\bar{\Gamma}'_f = \Gamma'_f[\mathtt{x}_{outer} \mapsto \Gamma_1(\mathtt{x}), \mathtt{this}_{outer} \mapsto \Gamma_1(\mathtt{this})]$$

This establishes, by the induction hypothesis on e_f, that in the state H', χ_1 the types $\Gamma_1(\mathtt{x})$ and $\Gamma_1(\mathtt{this})$ are still sufficient for the paths $\mathtt{x}$ and $\mathtt{this}$. Therefore, Γ_1 is sufficient for the state H', χ_1 as required.

The sufficiency $H', \chi', \Gamma' \vdash v : t_x \checkmark$ follows from the induction hypothesis on e_f and extends to $H', \chi', \Gamma' \vdash v : \mathcal{T}(f(e)) \checkmark$ due to $t_x \leq \mathcal{T}(f(e))$. □

Remark 6.6.4. Variance of the capacity

The proof case (T-MEMW∘) requires that all aliases of a type have the same capabilities. This restricts the typing rules in several cases: the rule (T-SWAP) cannot reduce the capacity of the resulting type (i.e. $(t, n_t - r/N_t - r)$) even though the sum of resource units stored with t has clearly been reduced and the rule (T-FUNX) requires $\mathcal{T}(e)_N =_N t_x$, instead of the expected $\mathcal{T}(e)_N \leq_N t_x$.

The additional restrictions on the capacities means that in certain cases the capacities and maybe even the reserved resource units included in types need to be over-approximated to allow a value to be passed as an argument or assigned to variable. Via a chain of related types, this could increase the capacities of multiple types. The ultimately leads to a bigger than necessary bound in the result. Therefore, a proof for the case (T-MEMW∘) without the requirement on the capacities could make the bounds more precise.

6.6.2.3 Differences to JavaScript

The difference between JavaScript handling and the function execution rule (S-FUNX) is that simple functions in JavaScript are called with the `this` variable set to the global scope object. Since AmorJiSe does not deal with scope interactions, `this` is set to be the empty object instead.

The decision not to concentrate on the handling of the scope in JavaScript is a real restriction, but is necessary to keep the rule manageable: In JavaScript functions usually have access to all variables defined in their own function scope (including the parameters), in addition to all variables accessible in the outer scope in which the function was defined in. To allow access to the outer scope in AmorJiSe would mean that the consumption of the function on the outer scope has to be monitored. To understand the difficulty of this consider the following example program.

```
1 function f1(x) {
2   var y = 5;
3   return function f2(x) {
4     CONSUME(y);
5   };
6 }
7 var f=f1(42);
8 f(43);
```

In this example, assume the function `CONSUME` requires a first parameter with the type $(\texttt{Int}, 1/1)$. The function `f1` defines the variable `y` and then returns the function

`f2`. Since `f2` is defined inside the scope of `f1`, it has access to the variable `y` and can use it as the parameter for `CONSUME`. In line 7, `f1` is called and returns a pointer to the newly created function `f2`. This new pointer is stored in the local variable `f`. When `f` is called later, it uses a resource unit stored with the variable `y`. At that time, `y` is not in scope anymore and AmorJiSe cannot reduce the annotations in the type of `y`.

Handling scopes in AmorJiSe is a challenge orthogonal to the main focus of AmorJiSe on the object structure of JavaScript. JavaScript functions have access to the scope of the defining context. With the previously discussed method of abstract locations for object types, the abstract location of the defining scope could simply be included in the function type and its type validated during the analysis of each function call. An easier approach is to set all annotations in the outer scope to 0 using the constraint t^0 in the rule (T-PROGRAM). This way the function body has access to the values of the outer scope without access to the reserved resource units stored in the types.

6.7 Implementation

The type inference algorithm discussed in Section 6.5 has been implemented[2] in Haskell using `ghc` in version 7.10.4. The implementation performs the following steps:

1. Parse the JavaScript code using the library `Language.JavaScript.Parser` (Version 0.5.14.7).

2. Infer the underlying types according to a re-implementation of the type system JS_0^T by Anderson.

3. Apply the AmorJiSe typing rules and collect the annotation constraints.

4. Solve the resulting linear programming problem using the Gnu Linear Programming Kit with its Haskell interface `glpk-hs` (Version: 0.3.5)

The *objective function* of the constructed LPP has to describe the potential of the initial environment. However, the LPP is constructed without knowledge of the runtime environment H, χ and without knowledge of the interaction sequence at runtime. For this reason, instead of the potential $\Sigma_H v : t$, the implementation uses a type potential $\Sigma_\Gamma t$, defined recursively as the sum of all annotations n occurring in the type t.

[2] Source code and instructions to run the system has been made available: `github.com/DFranzen/AmorJiSe`, a the docker image `dfranzen/amorjise` is available on `hub.docker.com`

Using this, for a type judgement $\Gamma, n \vdash P : t | \Gamma', n'$ the LPP minimises

$$n + 2 * \sum_{f \in P} \Sigma_{\Gamma'} \Gamma'(f).$$

In addition to the constantly required resource units, this minimises the resource units required by each function f defined in the program P to account for the interaction during runtime. Here, the potential cost of a function is weighted by a factor of 2 in comparison to the constant resource requirement n. This way, if there is a choice between the constant or interaction-dependent requirement, the constant requirement is preferred. This, however, does not effect the soundness of the bound, but is merely a heuristic to improve the quality of the resulting bound. Other objective functions might take different annotations into account and put different weights on the different annotations.

6.7.1 Extensions

The language analysed by the implementation is more expressive than the formal core handled by the theory above. At the highest level, the implementation handles expressions separate from statements and introduces statement types. Expressions are typed with the type provided above, while statements can be typed as `none` or `Return`(t) where t is an expression type to indicate that this statement returns a value of this type.

The typing rules for the discussed expression (including (T-SWAP) and function calls) are implemented as shown above. Additionally, the implementation extends the analysis of expressions with the following typing rules without formal proof:

The basic values of the analysed language have been extended by string and object literals via the rule (T-STRINGLIT) and (T-OBJLIT).

$$\frac{}{\Gamma, n + N + c_{strD} \vdash \text{"..."} : (\texttt{String}, N/N) | \Gamma, n} \quad \text{(T-STRINGLIT)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash (e_1, ..., e_k) : (t_1, ..., t_k) | \Gamma', n' \\ t_i \leq \mathcal{T}(e)(m_i) \\ \forall m \notin \{m_1, ..., m_k\} \mathcal{T}(e)(m)\, {}_n\!\leq_N \mathcal{T}(e)(m) \\ [\![\mathcal{T}(e)]\!]^n = N = [\![\mathcal{T}(e)]\!]^N\end{array}}{\Gamma, n + c_{objD} + k \cdot c_{memW} + N \vdash \underbrace{\{m_1 : e_1, ..., m_k : e_k\}}_{=e} : \mathcal{T}(e) | \Gamma', n'} \quad \text{(T-OBJLIT)}$$

The function handling has been extended to higher order functions, by allowing the called expression in $e(\cdot)$ to be an arbitrary expression e_f rather than just a function

name f. Additionally, the rule (FUNX) can type function calls with multiple parameters:

$$\frac{\begin{array}{c}\Gamma, n \vdash e_f : (\mathcal{T}(e_f), 0/0) | \Gamma_1, n_e \\ \mathcal{T}(e_f) = O \times (t_1, ..., t_k), n_f \to t_{ret}, n'_f \\ \Gamma_1, n_e \vdash (e_1, ..., e_k) : (\mathcal{T}(e_1), ..., \mathcal{T}(e_k)) | \Gamma_2, n_2 \\ O^0 \qquad (\mathcal{T}(e_1), ..., \mathcal{T}(e_k)) \leq (t_1, ..., t_k) \\ t_{ret} \leq \mathcal{T}(e_f(e_1, ..., e_k)) \\ n_f \leq n_2 \qquad n' = n_2 - n_f + n'_f\end{array}}{\Gamma, n + c_{funX} \vdash e_f(e_1, ..., e_k) : \mathcal{T}(e_f(e_1, ..., e_k)) | \Gamma_2, n'} \quad \text{(T-FUNX)}$$

This rule does not analyse polymorphic functions or variadicity. Instead, it restricts the function call to the exact number of parameters mentioned in the function definition.

In addition to normal function calls, the implementation includes specialised rules for method call:

$$\frac{\begin{array}{c}\Gamma, n \vdash e : (\mathcal{T}(e), 0/0) | \Gamma_1, n_e \\ \mathcal{T}(e)[m] = ((O \times (t_1, ..., t_k), n_f \to t_{ret}, n'_f, 0/0), \bullet) \\ \Gamma_1, n_e \vdash (e_1, ..., e_k) : (\mathcal{T}(e_1), ..., \mathcal{T}(e_k)) | \Gamma_2, n_2 \\ \mathcal{T}(e) \leq O \qquad (\mathcal{T}(e_1), ..., \mathcal{T}(e_k)) \leq (t_1, ..., t_k) \\ t_{ret} \leq \mathcal{T}(e.m(e_1, ..., e_k)) \\ n_f \leq n_2 \qquad n' = n_2 - n_f + n'_f\end{array}}{\Gamma, n + c_{funX} \vdash e.m(e_1, ..., e_k) : \mathcal{T}(e.m(e_1, ..., e_k)) | \Gamma_2, n'} \quad \text{(T-MEMX)}$$

The constructor call is handled just like a function call, since the receiver O of a function call is initialised without $\bullet$ fields. That means the constructor can be called with an empty object as `this`. During runtime the value $\{\}$ for `this` is provided by JavaScript.

$$\frac{\Gamma, n \vdash e_f(e) : t | \Gamma', n'}{\Gamma, n + c_{new} \vdash \texttt{new}\, e_f(e) : t | \Gamma', n'} \quad \text{(T-NEWX)}$$

The only difference of this rule to actual JavaScript behaviour is that in constructor calls which do not explicitly return a value, the value of `this` is returned. The rule (T-NEWX) does not capture this.

The syntax used for the functions here does not differentiate between the different kinds of function calls in JavaScript and allows the same function to be used in different roles throughout the code.

As last addition to the expression set, the implementation handles various operators in a rudimentary way. Standard arithmetic and logical operators require the operands to be of type `Int` or `Bool` and the (overloaded) plus operator requires both operands to be of equal data type.

The implementation also adds separate statement types $T^+ = (T, n/N)$. Only two kinds of statement pre-type T are considered: `none` or $\texttt{Return}(t)$ for any expression type t. The subtyping for those types is defined by the two rules

$$\texttt{Return}(t) \leq \texttt{none}$$

$$t \leq t' \Rightarrow \texttt{Return}(t) \leq \texttt{Return}(t')$$

Subtyping for full statement types T^+ is equivalent to expression types and the same holds true for the other type relations and constraints.

For the formal notation write the typing judgement for statements as $\Gamma, n \Vdash s : T|\Gamma', n'$. The implementation corresponds to typing rules as follows. Again they assume that the underlying system provides sound data-types for all parts of the analysed statement and only infer the constraints on the annotation variables. The implementation of the underlying system uses similar techniques to the expression typing to infer the types of the statements.

Any expression can also be typed as statements with the empty statement type:

$$\frac{\Gamma, n \vdash e : t|\Gamma', n'}{\Gamma, n \Vdash e : \texttt{none}|\Gamma', n'} \quad \text{(T-STMT)}$$

The return statement is typed with a return type containing the expression type of the returned expression.

$$\frac{\Gamma, n \vdash e : t|\Gamma', n'}{\Gamma, n \Vdash \texttt{return } e : \texttt{Return}(t)|\Gamma', n'} \quad \text{(T-RETURN)}$$

Equivalent to the rule for the conditional operator $e_1?e_2 : e_3$, the implementation includes a rule for the `if` statement.

$$\frac{\begin{array}{c}\Gamma, n \vdash e_b : \mathcal{T}(e_b)|\Gamma_1, n' \\ \Gamma_1, n' \Vdash s_t : \mathcal{T}(s_t)|\Gamma_t, n_t \\ \Gamma_1, n' \Vdash s_f : \mathcal{T}(s_f)|\Gamma_f, n_f \\ \mathcal{T}(s_t) \leq \mathcal{T}(\texttt{if...}) \qquad \mathcal{T}(s_f) \leq \mathcal{T}(\texttt{if...}) \\ \Gamma'\,{}_N{=}_N \uparrow(\Gamma_f, \Gamma_t) \qquad \Gamma'\,{}_n{=}_n \downarrow(\Gamma_t, \Gamma_f) \\ n'' = min(n_t, n_f)\end{array}}{\Gamma, n + c_{(\text{COND})} \Vdash \texttt{if}(e_b)\{s_t\}\ \texttt{else}\ \{s_f\} : \mathcal{T}(\texttt{if}(e_b)\{s_t\}\ \texttt{else}\ \{s_f\})|\Gamma', n''} \quad \text{(T-IF)}$$

Statement concatenation is typed with the following rule:

$$\frac{\begin{array}{c}\Gamma, n \Vdash s_1 : \mathcal{T}(s_1) | \Gamma_1, n' \\ \Gamma_1, n' \Vdash s_2 : \mathcal{T}(s_2) | \Gamma_2, n'' \\ \mathcal{T}(s_1) \leq \mathcal{T}(s_1; s_2) \qquad \mathcal{T}(s_2) \leq \mathcal{T}(s_1; s_2)\end{array}}{\Gamma, n + c_{(\text{SEQ})} \Vdash s_1; s_2 : \mathcal{T}(s_2) | \Gamma', n''} \quad \text{(T-STMTSEQ)}$$

The rule (T-PROGRAM) handling function definitions is split in the implementation. The rule (T-FUNSTMT) analyses function statements and (T-FUNEXPR) handles function expressions. The difference is mainly the mandatory function name in the function statement and the function type of the returned function value of the function expression.

$$\frac{\begin{array}{c}\mathcal{T}(f) = ((O \times (t_1, ..., t_k), n_f \to t_{ret}, n'_f), 0/0) \\ \Gamma_f = \Gamma[f \mapsto (\mathcal{T}(f), \bullet), \texttt{this} \mapsto O, (x_1, ..., x_k) \mapsto (t_1, ..., t_k)] \\ \Gamma_f, n_f \Vdash e : \mathcal{T}(e) | \Gamma_1, n'_f \\ t_{ret} = \begin{cases} \texttt{none} & \text{if } \mathcal{T}(e) = \texttt{none} \\ t & \text{if } \mathcal{T}(e) = return(t) \end{cases}\end{array}}{\Gamma, n + c_{(\text{FUND})} \vdash \texttt{function}[f](x_1, ..., x_k)\{e\} : \mathcal{T}(f) | \Gamma', n} \quad \text{(T-FUNEXPR)}$$

$$\frac{\begin{array}{c}\mathcal{T}(f) = ((O \times (t_1, ..., t_k), n_f \to t_{ret}, n'_f), 0/0) \\ \Gamma_f = \Gamma[f \mapsto (\mathcal{T}(f), \bullet), \texttt{this} \mapsto O, (x_1, ..., x_k) \mapsto (t_1, ..., t_k)] \\ \Gamma_f, n_f \Vdash e : \mathcal{T}(e) | \Gamma_1, n'_f \\ t_{ret} = \begin{cases} \texttt{none} & \text{if } \mathcal{T}(e) = \texttt{none} \\ t & \text{if } \mathcal{T}(e) = return(t) \end{cases}\end{array}}{\Gamma, n + c_{(\text{FUND})} \Vdash \texttt{function} f(x_1, ..., x_k)\{e\} : \texttt{none} | \Gamma', n} \quad \text{(T-FUNSTMT)}$$

The type of the function body has been adopted to return values marked with the `return` keyword instead of the last computed value. Another improvement is the ability to type nested functions. However, the scope chain has not been addressed. Nested function bodies can only access the variables defined within their own scope.

Finally, the analysed language is extended by the `var` keyword, which creates a new variable in the given context. To analyse these, the implementation emulates the three pass parsing of the JavaScript standard. In the first two passes over the source code only the initial typing context Γ_0 is populated with declared variables and functions using the rules (T-FUNSTMT) similar to (T-FUNEXPR) and the rule (T-VARD). This behaviour is equivalent to the runtime scope population in JavaScript. The third pass

actually analyses the given statement in this initial heap. Since the `var` keyword of a variable declaration `var` $x = e$ has already been analysed, in this third pass the variable declaration is treated as assignment $x = e$ instead. The semantics of the implementation requires variables to be declared in the scope they are used in and terminates with a warning, if an undeclared variable is assigned or read. The three pass procedure is repeated before every function body is analysed to discover variables local to its function scope.

The implementation uses equivalent rules to expand the type inference for the underlying type system JS_0^T. To convert the rules, the annotations are deleted and appropriate subtyping relations are inserted.

The layout of the analysis code is modular, such that a new rule can be added by supplying three functions. The first function receives an expression or statement as input and returns true, if the implemented rule applies; the second function breaks the expression into the relevant parts while the third function receives these parts and a typing environment and returns the type and the resulting changes to the environment. In this way new rules can be added easily.

6.7.2 Evaluation

The implementation has been tested on example code to verify the correct implementation of each rule.[3] Additionally, consider a simpler version of the Example 6.2.1:

```
1  function upload_list(list) {
2    UPLOAD(list.value);
3    upload_list(list.next)
4  }
```

For this example the resource model has been created to reflect the resource consumption of the API function `UPLOAD`:

```
1 "UPLOAD":(<(μ.({}),0/0),((Int,0/0))×(Int,0/0),0>,0/0)•,
```

With this type in the initial context, the analysis

```
1  ti_string "function upload_list(list) { UPLOAD(list.value);
       upload_list(list.next)}"
```

outputs the result

```
1  ...
```

[3]The test cases are part of the published implementation and can be run via the command ti_test

```
2 T(function upload_list)
3  (<(μβ_11.({}),0/0)×((μβ_13.({"value":(Int,0/0)•,"next":(μβ_16.({"
       value":(Int,0/0)•,"next":(β_16,1/1)•,}),1/1)•,}),0/1)),1->(
       NONE,0/0),0>,0/0)
4 whole program: 0 -> (NONE,0/0) ,0
```

It reveals that the parameter type for the function `upload_list` is an unfolding of the expected type

$$\mu\alpha.[\texttt{value} : ((\texttt{Int}, 0/0), \bullet), \texttt{next} : ((\alpha, 1/1), \bullet)].$$

The annotations $(1/1)$ show that one resource unit is required for each element in the list. For this example the LPP contained 146 linear constraints.

In general, the size of the generated LPP is difficult to estimate. Each applied rule generates a small number of linear or higher level constraints. However each higher level constraint is translated into a number of linear constraints potentially cubic in the size of the compared types. The relation between the size of the source code and the size of the resulting types is not clear. The rules for the `if` statement and the conditional operator (`e_bool?e_true:e_false`) compare whole typing contexts which results in linear constraints for all types in the typing context. This adds another unknown parameter to the size of the LPP.

Before this implementation can be applied to real world code, further constructs of the JavaScript language have to be added. To evaluate which constructs are most often missing, the parser and case-distinction was applied to the source code from the available 8000 PhoneGap apps. The test reported the loops and the array syntax used in most applications, but not analysed by AmorJiSe. Loops can be handled with a rule like the following:

$$\frac{\begin{array}{c}\Gamma, n \vdash e : \mathcal{T}(e_1) | \Gamma_1, n_1 \\ \Gamma_1, n_1 \Vdash e_2 : \mathcal{T}(e_2) | \Gamma_2, n_2 \\ \Gamma_2 \leq \Gamma \\ n_2 \geq n \\ \mathcal{T}(e_2) \leq \mathcal{T}(\texttt{while } (e_1)\{e_2\})\end{array}}{\Gamma, n + c_{while} \Vdash \texttt{while } (e_1)\{e_2\} : \mathcal{T}(\texttt{while } (e_1)\{e_2\}) | \Gamma_1, n_1} \quad \text{(T-WHILE)}$$

The analysis of Arrays could be handled similar to objects. The array type only needs a way to represent an arbitrary number of elements. For this, the δ field could potentially be adopted for amortised annotations.

During the conducted evaluation, a missing case in the JavaScript parser library was found and reported. It is fixed in the newest version.

6.8 Discussion

This chapter presented the formal type system AmorJiSe, which can be used to infer a bound on the resource consumption of JavaScript code. The system covers a core of JavaScript which is focused on the JavaScript specific object behaviour. The object types incorporate markers to allow controlled strong updates of object values by extension. For this core language, AmorJiSe automatically infers amortised annotations for provided data types. With those annotations, a bound for the resource behaviour can be inferred for the given JavaScript code, which can be data-dependent on the size of the input data and interaction dependent on the number of triggered events. Furthermore, the system has been implemented and extended in Haskell and applied to small examples.

AmorJiSe is formally proven sound in the sense that no execution can exceed the resulting bound on the resource usage. However, the soundness proof for AmorJiSe assumes the soundness of the provided data types. The combined system is only as sound as the underlying system. One might consider an unsound system with AmorJiSe to enhance the applicability and precision by sacrificing soundness in corner cases.

With the presented method, AmorJiSe can analyse API activated resources and language activated resources. The examples only used memory space as language activated resources, but with a more detailed resource model other such resources, like time or power consumption, can also be (over-)approximated with more detailed resource models. With respect to the resource classification in Section 2.2.1, AmorJiSe can analyse all four resource classes count, cumulative, unique and acquire-release. Even the uniqueness property of blocking resources can be modelled by adding the constraint $n \leq 1$ to the LPP for all annotations n in the typing judgement. However, with this adaption, no restriction can be posed on other apps. Therefore, blocking resources cannot be handled. Otherwise, AmorJiSe is only missing a method to handle resources which are defined via custom events like `lowbattery` or `online`.

The derived bounds can certainly be made more precise. Especially the rules (T-MEMW) and (T-VARW) overwrite values without considering the reserved resource units associated with the overwritten value. Instead, they could be extracted and restored into the globally available resource pool.

In some cases making the bounds more precise is more difficult. For example in a binary search trough a tree, the analysis would have to assume about one resource unit in each node of the tree. To reduce this number to the more precise resource unit per level of the tree, the analysis would have to be able to read resource units from other branches of the tree, which might not be possible in general. Similarly, amortised analysis can only use its full potential with recursive data-structures. Recursive functions which reduce an index parameter instead of the size of its data structure cannot be analysed with a non-zero resource consumption. Similar functions have been analysed in previous work by representing the decreasing numeric parameter as list of the appropriate length. This translation is, however, complicated and not equivalent in time and memory space. However, within those restrictions amortised analysis has the advantage that it circumvents dependent types and can therefore be efficiently inferred.

The version of AmorJiSe published as [39] did not use the heap-dependent definition of the potential Σ, yet. This, unfortunately, made the system unsound. This flaw has been corrected in the version presented here with the inclusion of the capacity annotations N.

The system is not yet usable on real world JavaScript apps. The language analysed needs to be extended with some more language constructs of JavaScript. Strongly related to the object analysis performed in AmorJiSe is the prototypal inheritance of JavaScript, which has not been included in the covered language. Prototypes make the object lookup highly dependent on the actual runtime state. Therefore, abstract types, which over-approximate all possible runtime states, become more complicated for prototypal inheritance. Prototypes have been handled by more complicated object lookup functions in previous research. While I expect that these results can be adapted to the inference of AmorJiSe, it would make the object lookup more ambiguous and consequently decrease precision of the types and bounds.

Concerning functions, the theoretic core of AmorJiSe restricts the functions to use exactly one variable with the identifier `x`. The implementation already relaxes this and allows functions to have an arbitrary amount of variables with arbitrary identifiers and even local variables. Still missing are nested function scopes and other scope manipulating language features. The type context is already equivalent to an object type. Therefore, scope objects in JavaScript can be typed using object types. The appropriate scope object type would then be embedded into each function type. This method naturally adds aliasing for the scope objects as a challenge. This challenge could be solved equivalent to the object aliasing proposed in this chapter by using the

sharing relation. This would dedicate a certain amount of resource units of the defining scope to be used exclusively inside the function scope. These blocked resource units might result in a less precise bound. Otherwise, the scope aliasing can be solved by indirect type pointers into an typing heap. This would, however, be a major alteration of the system.

Another part of JavaScript which is not included in AmorJiSe are labelled jumps using `break` and error handling. It does not interfere with the resource consumption heavily. Therefore, adaptations of the conventional methods are expected to suffice.

A rudimentary rule to handle `while` loops has been presented above. All other loops (`for`, `for each`, `until`) can be handled in a similar way. However, the bounds obtained from this handling are not very precise and would benefit from other known loop analysis techniques like loop unrolling and constant propagation. Especially in the case of `for` loops, shape analysis and other information about how often the loop is executed is vital for more precise bounds. Arrays are also left out of the analysed language. They could be modelled as objects with a δ field to describe arbitrary many of the indexed fields.

The challenge of dynamically executed JavaScript code using the `eval` function and others has not been addressed in this system. However, recent work as shown earlier suggests that most dynamic code in modern apps can be rewritten to static code. Those solutions are orthogonal to the type system presented here and could be added.

The implementation could benefit from a better description of the resource bounds. As it stands now the output only contains the typing judgement for the analysed expression. The types of the variables in the initial and final typing context can also be displayed optionally. From this output the bounds can be derived as described in Section 6.5.2. An explicit data dependent and interaction dependent bound is not provided by the system.

Lastly, in order to analyse real world JavaScript code AmorJiSe needs to handle additional libraries on top of JavaScript. Most notable is the DOM library which lets JavaScript interact with the current state of the HTML tree. Some parts of the DOM library have been handled in the example code for the evaluation by adding types of the relevant functions to the resource model. With the same method, the remaining parts of the DOM library and other libraries could be added to the analysed language. This avoids the blowup in complexity of adding the JavaScript code of the included libraries to the analysed code base.

Chapter 7

On the soundness of JS_0^T

The following chapter discusses a flaw in the soundness theorem (Theorem 1 in [9]) of the system JS_0^T. First this chapter will present an expression and a specially crafted state which invalidates one of the steps in the proof for the addressed theorem and formally show the validity of all needed preconditions for this counter example. The second part of the chapter shows that this crafted stated is actually reachable by providing a program which first generates this state from the empty state and then uses the flaw to produce a run-time error, even though the program can be typed as `Int`. This contradicts the type soundness of JS_0^T directly.

In [9] the author admits that the soundness proof of the type inference is not fully sound. Conjecture 3 in [9] states that the result computed from the closed constraints set is a correct solution to the constraints, but the proof attempt provided in the appendix is based, as noted by the author, on two unproven conjectures. The flaw presented here effects, instead, the soundness result for the type checking rules, which was, to the best of my knowledge, not known to be unsound so far. Upon contacting the authors, they failed to provide a correction to the handling of the counter example provided below.

A second flaw was discovered in the Theorem addressing the relation between type checking and type inference (Theorem 2 in [9]). In addition to presenting a counter example for this theorem, the last part of this chapter shows how the flawed inference rule for type checking can be corrected imitating the corresponding type inference rule. This addresses both presented issues. The modified typing rule is presented together with a proof for the flawed step in the soundness proof for type checking.

This chapter references definitions, theorems and proofs from the most recent publication of JS_0^T [9]. The numbering of the theorems has been adapted to fit the format

of this document, but the original numbers are provided with the theorem statements for reference. However, the same counter examples also apply to the corresponding theorems in [8], presented in ECOOP.

7.1 Error in the soundness proof

The main theorem for the type soundness of JS_0^T is repeated as Theorem 7.1.1 below. The theorem uses the relations for typing $\vdash$ (Figure 7.3), evaluation $\twoheadrightarrow$ (Figure 7.4), agreement $\diamond, \triangleleft$ (Figure 7.2) and extension $\sqsubseteq$ (Definition 7.1.6). For the full set of rules for evaluation and type checking, including those not interesting for the following examples, consult [9].

Theorem 7.1.1 ([9, Theorem 1,p. 64])**.** *Type Soundness of* JS_0^T.
Given

$$\begin{aligned} \mathbb{P} &: \textit{well-formed program} \\ t' &: \textit{Type} \\ e, w &: \textit{Expression} \end{aligned} \qquad \begin{aligned} \Gamma, \Gamma' &: \textit{Environments} \\ \mathcal{T} &: \textit{Store typings} \\ H, H' &: \textit{Heaps} \\ \chi, \chi' &: \textit{Stacks} \end{aligned}$$

with the preconditions

$$\begin{aligned} \mathbb{P}, \Gamma &\vdash e : t' | \Gamma' \\ \mathbb{P}, \Gamma, \mathcal{T} &\vdash H, \chi \diamond \\ e, H, \chi &\twoheadrightarrow w, H', \chi' \end{aligned}$$

there exists some $\mathcal{T}$*' with*

$$\begin{aligned} \mathbb{P}, \Gamma', \mathcal{T}' &\vdash H', \chi' \diamond \\ \mathcal{T}' &\sqsubseteq \mathcal{T} \end{aligned}$$

and either $\left\{ \begin{array}{l} w = v \textit{ with} \\ \mathbb{P}, H', \mathcal{T}' \vdash v \triangleleft t' \end{array} \right\}$ *or* $w = \texttt{nullPntrExc}$

Apart from the agreement of the derived type and value $\mathbb{P}, H', \mathcal{T}' \vdash v \triangleleft t'$, the theorem also claims the agreement of the heap typing with the heap $\mathbb{P}, \Gamma', \mathcal{T}' \vdash H', \chi' \diamond$ as an invariant of the type derivation.

The proof for this theorem explicitly constructs the new heap typing $\mathcal{T}'$ from $\mathcal{T}$. The proof distinguishes cases on the basis of the evaluation rule applicable to e. The

rule (S-MEMASS) evaluates assignments to object members $e_1.m = e_2$ with expressions e_1, e_2 and in particular assignments of the special form $var.m = e_2$. The induction step of the proof for the expression case $var.m = e_2$ uses the induction assumption to derive the existence of a new heap typing $\mathcal{T}''$ after the evaluation of e_2. From $\mathcal{T}''$ the required heap typing $\mathcal{T}'$ is derived in line (26) as

> Now let:
> $$\mathcal{T}' = \mathcal{T}''[\iota \mapsto \mathcal{T}''[m \mapsto (t, \bullet)]]. \tag{26}$$

Here ι is the heap location of the variable *var* and t is the type derived for the expression e_2. To establish the extension relation $\mathcal{T}' \sqsubseteq \mathcal{T}$ required by the theorem, line (27) claims

> From (26) and the definition of $\sqsubseteq$ [see Def. 7.1.8] we get:
> $$\mathcal{T}' \sqsubseteq \mathcal{T}'' \tag{27}$$

which is not actually the case as will be shown in the following.

7.1.1 Counter example

This first counter example invalidates the agreement claim $\mathbb{P}, \Gamma', \mathcal{T}' \vdash H', \chi' \diamond$ for the explicitly constructed heap typing $\mathcal{T}'$. It consists of the expression

$$e := \texttt{x.val=3}$$

and the crafted state *CE*:

$$\begin{aligned}
\chi_{CE} : \{&\texttt{x} \mapsto \iota_x, \\
&\texttt{this} \mapsto \iota_t\}
\end{aligned}
\qquad
\begin{aligned}
\Gamma_{CE} : \{&\texttt{x} \mapsto O_\circ, \\
&\texttt{this} \mapsto []\}
\end{aligned}$$

$$\begin{aligned}
H_{CE} : \{&\iota_x \mapsto \ll \texttt{next} : \iota_n \gg, \\
&\iota_n \mapsto \ll \texttt{next} : \texttt{null} \gg, \\
&\iota_t \mapsto \ll\gg\}
\end{aligned}
\qquad
\begin{aligned}
\mathcal{T}_{CE} : \{&\iota_x \mapsto O_\circ, \\
&\iota_n \mapsto O_\circ, \\
&\iota_t \mapsto []\}
\end{aligned}$$

The types have been abbreviated for readability:

$$O_\circ := \mu\alpha[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)]$$
$$O_\bullet := \mu\alpha[\texttt{val} : (\texttt{Int}, \bullet), \texttt{next} : (\alpha, \bullet)]$$

Figure 7.1 Type Relations

$$\frac{t \equiv t'}{(t,\psi) \equiv (t',\psi)} \ (\equiv \text{MEMBER})$$

$$\frac{\forall m : M(m) \equiv M'(m)}{\mu\alpha.M \equiv \mu\alpha.M'} \ (\equiv \text{REORDER})$$

$$\frac{\exists t'' \quad t''[\alpha/t] \equiv t \qquad t''[\alpha/t'] \equiv t'}{t \equiv t'} \ (\equiv \text{FIX})$$

$$\frac{}{\mu\alpha.M \equiv \mu\alpha'.M[\alpha/\alpha']} \ (\equiv \alpha\text{CONV})$$

$$\frac{}{t \equiv t} \ (\equiv \text{REFLEX})$$

$$\frac{}{\mu\alpha.M \equiv M[\alpha/\mu\alpha.M]} \ (\equiv \text{UNFOLD})$$

(a) Congruence

$$\frac{\psi' = \bullet \Rightarrow \psi = \bullet}{\psi \leq \psi'} \ (\leq \text{ANO})$$

$$\frac{t \equiv t' \qquad \psi \leq \psi'}{(t,\psi) \leq (t',\psi')} \ (\leq \text{MEM})$$

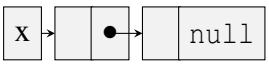
$$\frac{\forall m \in \mathtt{Dom}(O') : O(m) \leq O'(m)}{O \leq O'} \ (\leq \text{OBJ})$$

$$\frac{t \equiv t'}{t \leq t'} \ (\leq \text{CONG})$$

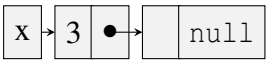
$$\frac{t \leq t' \qquad t' \leq t''}{t \leq t''} \ (\leq \text{TRANS})$$

(b) Subtypes

Intuitively in the presented heap `x` contains a recursive linked list with two elements.



The list is able to store integer values in the field `val` as the type $O_\circ$ suggests with `val` : (`Int`, `o`), but there are no values stored so far. In the expression `x.val=3`, a value is added to the head of the list.



Throughout this example, the variable `this` can be ignored, but its type and value is provided for formality.

7.1.2 Types for the counter example

The types of the fields of an object type can be read via the field lookup:

Definition 7.1.2 ([9, p. 48])**.** *Field lookup*

For a bound type $O = \mu\alpha.M$ with $M = [m_1 : (t_1, \psi_1), ..., m_k : (t_k, \psi_k)]$ field lookup is defined as

$$O(m) = M[\alpha/O](m)$$
$$M(m) = \begin{cases} (t_i, \psi_i) & \text{if } m = m_i \text{ for some } i \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

where $M[\alpha/O]$ is the standard non-recursive replacing of α by O in M.

The first step is to show that the two types $O_\bullet$ and $O_\circ$ are not comparable via the relations congruence $\equiv$ (Figure 7.1a), subtype $\leq$ (Figure 7.1b) and extension $\sqsubseteq$ (Definition 7.1.6). For this it is useful to have additional properties of the relation $\equiv$:

Proposition 7.1.3. *Properties of $\equiv$*

1. *The relation $\equiv$ is symmetric: given $t_1 \equiv t_2$ it holds that $t_2 \equiv t_1$.*
2. *The relation $\equiv$ is transitive: given $t_1 \equiv t_2$ and $t_2 \equiv t_3$ it holds that $t_1 \equiv t_3$.*

Proof. 1. Consider the rule ($\equiv$FIX) with $t'' = t = t_1$ and $t' = t_2$

$$\dfrac{\dfrac{t_1 \equiv t_2}{t_1[\alpha_{new}/t_2] \equiv t_2}\,(\text{SUBSTITUTE}) \qquad \dfrac{\dfrac{}{t_1 \equiv t_1}\,(\equiv \text{REFLEX})}{t_1[\alpha_{new}/t_1] \equiv t_1}\,(\text{SUBSTITUTE})}{t_2 \equiv t_1}\,(\equiv \text{FIX})$$

2. Choose an α not contained in t_2. With this choice it holds $t_2[\alpha/t_1] = t_2[\alpha/t_3] = t_2$. The fact $t_2 \equiv t_3$ is one of the requirements and the previous proof derives $t_2 \equiv t_1$ from $t_1 \equiv t_2$. The claim then follows from

$$\dfrac{t_2[\alpha/t_1](= t_2) \equiv t_1 \qquad t_2[\alpha/t_3](= t_2) \equiv t_3}{t_1 \equiv t_3}\,(\equiv \text{FIX})$$

□

These proof trees will be abbreviated as the rules ($\equiv$ SYM) and ($\equiv$ TRANS) in the following.

Now for the relation between $O_\bullet$ and $O_\circ$, the first relation to consider is $\equiv$, since $\leq$ and $\sqsubseteq$ both depend on $\equiv$.

Proposition 7.1.4.

1. *For two object types O, O', if*

$$\underbrace{\mu\alpha.[m_1 : (t_{m1}, \psi_{m1}), ..., m_k : (t_{mk}, \psi_{mk})]}_{O} \equiv \underbrace{\mu\alpha.[m'_1 : (t'_{m1'}, \psi'_{m1'}), ..., m'_{k'} : (t'_{mk'}, \psi'_{mk'})]}_{O'}$$

then $\{m_1, ..., m_k\} = \{m'_1, ..., m'_{k'}\}$ and $\psi_{mi} = \psi'_{mi}$ for all $1 \leq i \leq k$.

2. $O_\bullet \not\equiv O_\circ$

Proof. 1. This statement can be proven via Induction on the derivation of the $\equiv$ relation (see Figure 7.1).

Case ($\equiv$ REORDER): The preconditions of ($\equiv$ REORDER), are that all member types $O(m)$ are congruent. As a direct consequence, the sets $\{m_1, ..., m_k\}$ and $\{m'_1, ..., m'_{k'}\}$ have to be equal, otherwise only one of $O(m)$ and $O'(m)$ would be $\mathcal{U}df$ which is only congruent to $\mathcal{U}df$. The two members can either be congruent by the rule ($\equiv$ MEMBER) or ($\equiv$ FIX).

- According to the rule ($\equiv$ MEMBER), two member types (t, ψ) can only be congruent if the states ($\psi \in \{\bullet, \circ\}$) are identical. Therefore the states ψ_m have to be equal in both object types.
- If the rule ($\equiv$ FIX) was applied to deduce that the member types are congruent, then the derivation of $t''[\alpha/O] \equiv O$ and $t''[\alpha/O'] \equiv O'$ are shorter than the derivation of $O \equiv O'$ and the induction hypothesis can be applied. Since the substitutions $[\alpha/O']$ and $[\alpha/O]$ do not change the state ψ of the members, transitivity can be applied to get the equality of the states in O and O'.

Case ($\equiv$ FIX): The rule ($\equiv$ FIX) has $t \equiv t''[\alpha/t]$ and $t' \equiv t''[\alpha/t']$ as precondition. The induction hypothesis applies to those relations and the substitutions $[\alpha/t]$ and $[\alpha/t']$ do not change the set of members $\{m''_1, ..., m''_{k''}\}$ or the involved ψ''_{mi}. From this fact it follows that $\{m_1, ..., m_k\} = \{m''_1, ..., m''_{k''}\} = \{m'_1, ..., m'_{k'}\}$ and $\psi_{mi} = \psi''_{mi} = \psi'_{mi}$ as claimed.

Case ($\equiv$ αCONV): The set of fields $\{m_1, ..., m_k\}$ and the states $\psi_{m1}, ..., \psi_{mk}$ are trivially identical in the types $\mu\alpha.M$ and $\mu\alpha'.M[\alpha/\alpha']$.

Case ($\equiv$ UNFOLD): The set of fields $\{m_1, ..., m_k\}$ and the states $\psi_{m1}, ..., \psi_{mk}$ are trivially identical in the types $\mu\alpha.M$ and $M[\alpha/\mu\alpha.M]$.

2. Since for $O_\bullet$ and $O_\circ$ it holds $\psi_{val} = \bullet \neq \circ = \psi'_{val}$, this statement is a direct consequence of the previous proof by contradiction.

□

The object types are not *subtypes* either.

Proposition 7.1.5.

1. *For two object types in a subtyping relation $O \leq O'$ and each field m with $O(m) = (t_m, \psi_m)$ and $O'(m) = (t'_m, \psi'_m)$ the actual types t_m and t'_m are congruent.*
2. *$O_\circ \not\leq O_\bullet$ and $O_\bullet \not\leq O_\circ$*

Proof. 1. This statement can be shown by induction on the derivation for $O \leq O'$ (see Figure 7.1):

Case ($\leq$ OBJ):: This rule requires the subtyping relation for the types of each field m. By the subtyping rule ($\leq$ MEM), two member types (t_m, ψ), (t'_m, ψ') are only subtypes if the actual types t_m and t'_m are congruent.

Case ($\leq$ CONG):: The requirement of this rule is $t \equiv t'$ which obviously requires each field to be congruent by the definition of $\equiv$.

Case ($\leq$ TRANS):: This rule requires the existence of an additional type O'' with $O \leq O''$ and $O'' \leq O'$. By the induction hypothesis, all field types of O are congruent to the field types in O''. Equally, by the induction hypothesis O'' has equivalent field types to O'. The claim follows directly by the transitivity of $\equiv$ as proven in Proposition 7.1.3.

2. Follows from the previous proof by contradiction since the types of field `next` are by the definition of field lookup (Definition 7.1.2).

$$\begin{aligned} O_\circ(\texttt{next}) &= (\alpha[\alpha/O_\circ], \bullet) \\ &= (O_\circ, \bullet) \\ O_\bullet(\texttt{next}) &= (\alpha[\alpha/O_\bullet], \bullet) \\ &= (O_\bullet, \bullet) \end{aligned}$$

and Proposition 7.1.4 established that $O_\bullet \not\equiv O_\circ$.

□

Finally, consider the *extension* relation $\sqsubseteq$:

Definition 7.1.6 ([9, Definition 2, p. 57])**.** *Extension*

$O_\bullet \sqsubseteq O_\circ$ *iff for all m:*

- $O_\circ(m) = \mathcal{U}df \Leftrightarrow O_\bullet(m) = \mathcal{U}df$
- $O_\circ(m) \neq \mathcal{U}df \Rightarrow O_\bullet(m) \leq O_\circ(m)$

Proposition 7.1.7.

$O_\bullet \not\sqsubseteq O_\circ$ *and* $O_\circ \not\sqsubseteq O_\bullet$.

Proof. Using the definition of the field lookup for the field $m = \texttt{next}$ again, the second condition requires $(O_\bullet, \bullet) \leq (O_\circ, \bullet)$, which is not true as already discussed above. Therefore $O_\bullet \not\sqsubseteq O_\circ$. The argument works equivalently for $O_\circ \not\sqsubseteq O_\bullet$. □

This concludes the discussion of the relation between $O_\circ$ and $O_\bullet$ with the result that the two types are not related via the given relations.

7.1.3 Preconditions of the counter example

In order to show that the expression `x.val=3` in the given state $H_{CE}, \chi_{CE}, \Gamma_{CE}, \mathcal{T}_{CE}$ is a counter example for Theorem 7.1.1, the following presents derivations of all preconditions of Theorem 7.1.1.

7.1.3.1 Agreement

The agreement relation asserts that the types in the context Γ_{CE} and the heap typing $\mathcal{T}_{CE}$ are adequate types for the values stored in the stack χ_{CE} and the heap H_{CE}. The definition of this relation $\Gamma_{CE}, \mathcal{T}_{CE} \vdash H_{CE}, \chi_{CE} \diamond$ is repeated in Figure 7.2. The rules are syntax directed and therefore there is only one possible derivation for each such statement.

In the given situation the rule (A-WLFHEAPSTACK) applies and reduces the agreement to the following preconditions:

- $\texttt{Dom}(\mathcal{T}_{CE}) = \{\iota_x, \iota_t, \iota_n\} = \texttt{Dom}(H_{CE})$ ✓
- $H_{CE}, \mathcal{T}_{CE} \vdash \chi_{CE}(\texttt{x}) \triangleleft \Gamma_{CE}(\texttt{x})$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{O_\circ \equiv O_\circ}\,(\equiv \text{REFLEX})}{O_\circ \leq O_\circ}\,(\leq \text{CONG})}{\mathcal{T}_{CE} \vdash \iota_x \prec O_\circ}\,(\text{A-WEAKADDR}) \quad \dfrac{\dfrac{\dfrac{}{O_\circ \equiv O_\circ}\,(\equiv \text{REFLEX})}{O_\circ \leq O_\circ}\,(\leq \text{CONG})}{\mathcal{T}_{CE} \vdash \iota_n \prec O_\circ}\,(\text{A-WEAKADDR})}{H_{CE}, \mathcal{T}_{CE} \vdash \iota_x \triangleleft \mu\alpha.[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)]}\,(\text{A-STRONGADDR})$$ ✓

Figure 7.2 Definition: Agreement

$$\frac{}{\mathbb{P},\mathcal{T} \vdash n \prec \texttt{Int}} \quad \text{(A-WEAKINT)}$$

$$\frac{}{\mathbb{P},\mathcal{T} \vdash \texttt{null} \prec \texttt{null}} \quad \text{(A-WEAKNULL)}$$

$$\frac{\mathbb{P}(f) = \texttt{function } f(\texttt{x}) : G'\{\ldots\} \quad G \equiv G'}{\mathbb{P},\mathcal{T} \vdash f \prec G} \quad \text{(A-WEAKFUNC)}$$

$$\frac{\mathcal{T}(\iota) \leq O}{\mathbb{P},\mathcal{T} \vdash \iota \prec O} \quad \text{(A-WEAKADDR)}$$

$$\frac{\begin{array}{c}\mathbb{P},\mathcal{T} \vdash \iota \prec O \\ H(\iota) = \ll m_1 : v_1, \ldots, m_p : v_p \gg \\ \forall m : O(m) = (t,\bullet) \Rightarrow \exists i \in 1\ldots p : m = m_i \wedge \mathbb{P},\mathcal{T} \vdash v_i \prec t \\ \forall i \in 1\ldots p : O(m_i) = (t,\circ) \Rightarrow \mathbb{P},\mathcal{T} \vdash v_i \prec t\end{array}}{\mathbb{P},H,\mathcal{T} \vdash \iota \triangleleft O} \quad \text{(A-STRONGADDR)}$$

$$\frac{\begin{array}{c}\texttt{Dom}(\mathcal{T}) = \texttt{Dom}(H) \\ \forall \iota : \mathcal{T}(\iota) = O \Rightarrow \mathbb{P},H,\mathcal{T} \vdash \iota \triangleleft O \\ \mathbb{P},H,\mathcal{T} \vdash \chi(\texttt{this}) \triangleleft \Gamma(\texttt{this}) \\ \mathbb{P},H,\mathcal{T} \vdash \chi(\texttt{x}) \triangleleft \Gamma(\texttt{x})\end{array}}{\mathbb{P},\Gamma,\mathcal{T} \vdash H,\chi \diamond} \quad \text{(A-WLFHEAPSTACK)}$$

- $H_{CE},\mathcal{T}_{CE} \vdash \chi_{CE}(\texttt{this}) \triangleleft \Gamma_{CE}(\texttt{this})$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[] \equiv []}\,(\equiv \text{REFLEX})}{[] \leq []}\,(\leq \text{CONG})}{\mathcal{T}_{CE} \vdash \iota_t \prec []}\,\text{(A-WEAKADDR)}}{H_{CE},\mathcal{T}_{CE} \vdash \iota_t \triangleleft []}\,\text{(A-STRONGADDR)} \qquad \checkmark$$

- $\forall \iota' \in \texttt{Dom}(\mathcal{T}_{CE})$ with $\mathcal{T}_{CE}(\iota') = O' : H_{CE},\mathcal{T}_{CE} \vdash \iota' \triangleleft O'$

  The cases for $\iota' = \iota_t$ and $\iota' = \iota_x$ have already been handled. That leaves ι_n

$$\dfrac{\dfrac{\dfrac{\dfrac{}{O_\circ \equiv O_\circ}\,(\equiv \text{REFLEX})}{O_\circ \leq O_\circ}\,(\leq \text{CONG})}{\mathcal{T}_{CE} \vdash \iota_n \prec O_\circ}\,\text{(A-WEAKADDR)} \quad \dfrac{}{\mathcal{T}_{CE} \vdash \texttt{null} \prec O_\circ}\,\text{(A-WEAKNULL)}}{H_{CE},\mathcal{T}_{CE} \vdash \iota_n \triangleleft \mu\alpha.[\texttt{val} : (\texttt{Int},\circ), \texttt{next} : (\alpha,\bullet)]}\,\text{(A-STRONGADDR)} \qquad \checkmark$$

7.1.3.2 Typing

The typing relation $\mathbb{P},\Gamma \vdash e : t' | \Gamma'$ asserts that in the context Γ the expression (here $e =$`x.val=3`) can be typed with the type t' (here $t' =$ `Int`). The relation is defined via

Figure 7.3 Definition: Type checking

$$\frac{}{\mathbb{P},\Gamma \vdash n : \texttt{Int}} \text{(T-CONST)} \qquad \frac{}{\mathbb{P},\Gamma \vdash \texttt{this} : \Gamma(\texttt{this})} \text{(T-VAR)}$$

$$\mathbb{P},\Gamma \vdash \texttt{x} : \Gamma(\texttt{x})$$

$$\frac{\begin{array}{c}\mathbb{P},\Gamma \vdash e_1 : O|\Gamma' \\ \mathbb{P},\Gamma' \vdash e_2 : t|\Gamma'' \\ O(m) = (t'', \bullet) \\ t \leq t''\end{array}}{\mathbb{P},\Gamma \vdash e_1.m = e_2 : t|\Gamma''} \text{(T-ASSIGNUPD)}$$

$$\frac{\begin{array}{c}\mathbb{P},\Gamma \vdash e_2 : t|\Gamma' \\ \texttt{var=}e' \text{ is not a subexpression of } e_2 \\ \Gamma'(\texttt{var}) = O \\ O(m) = (t'', \psi) \\ t \leq t'' \\ \Gamma'' = \Gamma'[\texttt{var} \mapsto O[m \mapsto (t'', \bullet)]]\end{array}}{\mathbb{P},\Gamma \vdash \texttt{var}.m = e_2 : t|\Gamma''} \text{(T-ASSIGNADD)}$$

$$\frac{\begin{array}{c}\mathbb{P},\Gamma \vdash e : t|\Gamma' \\ \mathbb{P}(f) = \texttt{function } f(\texttt{x}) : G\{\ldots\} \\ t \leq G(\texttt{x}) \\ G(\texttt{this}) \text{ has no } \bullet \text{ fields}\end{array}}{\begin{array}{c}\mathbb{P},\Gamma \vdash \texttt{new } f(e) : G(ret)|\Gamma' \\ \mathbb{P},\Gamma \vdash f(e) : G(ret)|\Gamma'\end{array}} \text{(T-CALL)}$$

$$\frac{\mathbb{P},\Gamma \vdash e_1 : t|\Gamma' \qquad \mathbb{P},\Gamma \vdash e_2 : t'|\Gamma''}{\mathbb{P},\Gamma \vdash e_1; e_2 : t'|\Gamma''} \text{(T-SEQ)}$$

inference rules. The rules needed for this example are summarised in Figure 7.3

According to those rules, the example `x.val=3` can be typed as `Int`

$$\frac{\begin{array}{c}\frac{}{\Gamma_{CE} \vdash 3 : \texttt{Int}|\Gamma'_{CE}} \text{(T-CONST)} \\ \texttt{x=e}' \text{ is not a sub-expression of } 3 \\ \Gamma'_{CE}(\texttt{x}) = O_\circ \\ O_\circ(\texttt{val}) = (\texttt{Int}, \circ) \\ \texttt{Int} \leq \texttt{Int} \\ \Gamma''_{CE} = \Gamma'_{CE}[\texttt{x} \mapsto \Gamma'_{CE}(x)[\texttt{val} \mapsto (\texttt{Int}, \bullet)]]\end{array}}{\Gamma_{CE} \vdash \texttt{x.val=3} : \texttt{Int}|\Gamma''_{CE}} \text{(T-ASSIGNADD)}$$

7.1.3.3 Evaluation

The big-step evaluation relation $e, H, \chi \twoheadrightarrow w, H', \chi'$ asserts that in the environment H, χ the expression e can be evaluated to the value w. The evaluation is defined using the inference rules repeated in Figure 7.4.

According to those rules, the expression `x.val=3` can be evaluated to the value 3 which agrees with the behaviour of real world JavaScript engines and the typing as `Int`.

Figure 7.4 Definition: Evaluation

$$\frac{}{\mathtt{x},H,\chi \twoheadrightarrow \chi(\mathtt{x}),H,\chi}\ (\text{S-VAR}) \qquad \frac{}{n,H,\chi \twoheadrightarrow n,H,\chi}\ (\text{S-VAL})$$

$$\frac{e,H,\chi \twoheadrightarrow \iota,H',\chi'}{e.m,H,\chi \twoheadrightarrow H'(\iota)(m),H',\chi'}\ (\text{S-MEMSEL}) \qquad \frac{\begin{array}{c}e_1,H,\chi \twoheadrightarrow \iota,H_1,\chi_1\\ e_2,H_1,\chi_1 \twoheadrightarrow v,H_2,\chi'\\ H' = H_2[\iota \mapsto H_2(\iota)[m \mapsto v]]\end{array}}{e_1.m = e_2,H,\chi \twoheadrightarrow v,H',\chi'}\ (\text{S-MEMASS})$$

$$\frac{\begin{array}{c}e,H,\chi \twoheadrightarrow v',H1,\chi'\\ P(f) = \mathtt{function}\ f(\mathtt{x})\{e'\}\\ e',H_1,\{\mathtt{this} \mapsto \mathtt{null},\mathtt{x} \mapsto v'\} \twoheadrightarrow v,H',\chi''\end{array}}{f(e),H,\chi \twoheadrightarrow v,H',\chi'}\ (\text{S-FUNCCALL})$$

$$\frac{\begin{array}{c}e,H,\chi \twoheadrightarrow v',H_1,\chi'\\ P(f) = \mathtt{function}\ f(\mathtt{x})\{e'\}\\ \iota \text{ is new in } H_1 \text{ and } H_2 = H_1[\iota \mapsto \ll\gg]\\ e',H_2,\{\mathtt{this} \mapsto \iota,\mathtt{x} \mapsto v'\} \twoheadrightarrow v,H',\chi''\end{array}}{\mathtt{new}\ f(e),H,\chi \twoheadrightarrow \iota,H',\chi'}\ (\text{S-NEW})$$

$$\frac{\begin{array}{c}e_1,H,\chi \twoheadrightarrow v',H_1,\chi_1\\ e_2,H_1,\chi_1 \twoheadrightarrow v,H',\chi'\end{array}}{e_1;e_2,H,\chi \twoheadrightarrow v,H',\chi'}\ (\text{S-SEQ}) \qquad \frac{\begin{array}{c}e,H,\chi \twoheadrightarrow \iota,H',\chi'\\ H'(\iota)(m) = \mathcal{U}df\end{array}}{e.m,H,\chi \twoheadrightarrow \mathtt{stuckErr},H',\chi'}\ (\text{S-NOMEM})$$

$$\frac{}{\mathtt{x},H_{CE},\chi_{CE} \twoheadrightarrow \iota_x,H_{CE,1},\chi_{CE,1}}\ (\text{S-VAR}) \qquad \frac{}{3,H_{CE,1},\chi_{CE,1} \twoheadrightarrow 3,H_{CE,2},\chi'_{CE}}\ (\text{S-VAL})$$

$$\frac{H'_{CE} = H_{CE,2}[\iota_x \mapsto H_{CE,2}(\iota_x)[\mathtt{var} \mapsto 3]]}{\mathtt{x.val{=}3},H_{CE},\chi_{CE} \twoheadrightarrow 3,H'_{CE},\chi'_{CE}}\ (\text{S-MEMASS})$$

7.1.3.4 Post-state

The given derivation in the provided starting state $\chi_{CE},H_{CE},\Gamma_{CE},\mathcal{T}_{CE}$ results in the following intermediate and post-states:

$$\begin{aligned} H'_{CE} = {} & \{\iota_x \mapsto \ll \mathtt{next} : \iota_n, \mathtt{val} : 3 \gg \\ & \iota_n \mapsto \ll \mathtt{next} : \mathtt{null} \gg, \\ & \iota_t \mapsto \ll\gg\}, \\ \Gamma''_{CE} = {} & \{\mathtt{x} \mapsto \mu\alpha.[\mathtt{val} : (\mathtt{Int},\bullet), \mathtt{next} : (\alpha,\bullet)], \\ & \mathtt{this} \mapsto []\} \end{aligned}$$

$$\chi'_{CE} = \chi_{CE,1} = \chi_{CE}$$

$$H_{CE,1} = H_{CE,2} = H_{CE}$$

$$\Gamma'_{CE} = \Gamma_{CE}$$

The proof of the soundness for the case of e uses the soundness of the typing $\mathbb{P},\Gamma_{CE} \vdash 3 : \mathtt{Int} | \Gamma'_{CE}$ as induction hypothesis. Due to this induction hypothesis, there exists a new heap typing $\mathcal{T}''_{CE}$, for which the agreement $\mathbb{P},\Gamma'_{CE},\mathcal{T}''_{CE} \vdash H_{CE,2},\chi'_{CE} \diamond$ holds. Since the expression 3 does not change the state, i.e. $H_{CE,2} = H_{CE},\chi'_{CE} = \chi_{CE},\Gamma'_{CE} = \Gamma_{CE}$, the heap typing $\mathcal{T}''_{CE}$ in the induction hypothesis is chosen to be $\mathcal{T}''_{CE} = \mathcal{T}_{CE}$.

This concludes the discussion of the preconditions for the counter example. Since they all hold true for the state and expression of the counter example, Theorem 1 [9] is applicable.

7.1.4 Wrong conclusion

The two important lines in the proof for the Theorem 7.1.1 are:

> Now let:
> $$\mathcal{T}' = \mathcal{T}''[\iota \mapsto \mathcal{T}''(\iota)[m \mapsto (t, \bullet)]] \tag{26}$$
> From (26) and the definition of $\sqsubseteq$ we get:
> $$\mathcal{T}' \sqsubseteq \mathcal{T}'' \tag{27}$$

Applied to the counter example *CE* this specialises to:

$$\mathcal{T}'_{CE} = \mathcal{T}''_{CE}[\iota_x \mapsto \mathcal{T}''_{CE}(\iota_x)[\mathtt{var} \mapsto (\mathtt{Int}, \bullet)]]$$

and the result is the following concrete heap typing:

$$\begin{aligned}\mathcal{T}'_{CE} = \{&\iota_x \mapsto \mu\alpha.[\mathtt{val} : (\mathtt{Int}, \bullet), \mathtt{next} : (\alpha, \bullet)]\\ &\iota_n \mapsto \mu\alpha.[\mathtt{val} : (\mathtt{Int}, \circ), \mathtt{next} : (\alpha, \bullet)]\\ &\iota_t \mapsto []\}\end{aligned}$$

The relation $\sqsubseteq$ in line (27) is the extension of $\sqsubseteq$ for heap typings which is defined as:

Definition 7.1.8 ([9, Definition 2,p. 61])**.** *We say that $\mathcal{T}'$ extends $\mathcal{T}''$ denoted by $\mathcal{T}' \sqsubseteq \mathcal{T}''$ iff,*

- $\mathtt{Dom}(\mathcal{T}'') \subseteq \mathtt{Dom}(\mathcal{T}')$ *and*
- $\mathcal{T}''(\iota) = O \Rightarrow \mathcal{T}'(\iota) \sqsubseteq O$ *(for all addresses ι)*

In particular, it requires $O_\bullet = \mathcal{T}'(\iota_x) \sqsubseteq \mathcal{T}''(\iota_x) = O_\circ$ which was shown incorrect in Proposition 7.1.7. Therefore, line (27) is not correct and the claim

$$\mathcal{T}' \sqsubseteq \mathcal{T}$$

in the results of Theorem 7.1.1 is not correct in *CE* for the constructed $\mathcal{T}'$. In addition the agreement

$$\mathbb{P}, \Gamma', \mathcal{T}' \vdash H', \chi' \diamond$$

also claimed by the theorem is invalid as it would require $O_\circ \leq O_\bullet$:

$$\dfrac{\dfrac{\dfrac{}{O_\bullet \equiv O_\bullet}\,(\equiv \text{REFLEX})}{O_\bullet \leq O_\bullet}\,(\leq \text{EQUIV})}{\mathcal{T}'_{CE} \vdash \iota_x \prec O_\bullet}\,(\text{A-WEAKADDR}) \qquad \dfrac{}{\mathcal{T}'_{CE} \vdash 3 \prec \texttt{Int}}\,(\text{A-WEAKINT}) \qquad \dfrac{\lightning}{O_\circ \leq O_\bullet}\,(\text{A-WEAKADDR})}{H'_{CE}, \mathcal{T}'_{CE} \vdash \iota_x \lhd \mu\alpha.[\texttt{val} : (\texttt{Int}, \bullet), \texttt{next} : (\alpha, \bullet)]}\,(\text{A-STRONGADDR})$$

For this simple counter example, a different (less expressive) $\mathcal{T}'$ could be constructed which validates the agreement. However, the following section addresses that the example *CE* can be exploited to invalidate Theorem 7.1.1 independent of the choice of $\mathcal{T}'$.

7.1.5 Full counter example

The previous section showed the wrong conclusion in the proof of the soundness theorem. This flaw results in a situation with inconsistent heap typing $\Gamma'', \mathcal{T}' \not\vdash H', \chi' \diamond$. This section extends the counter example above to a full JavaScript program which is typed as `Int` but correctly returns a run-time error $w = \texttt{stuckError}$. This clearly contradicts the claim of type soundness. Since the type inference soundness has the additional precondition of an empty starting heap, this full counter example also shows that the crafted state $H_{CE}, \chi_{CE}, \Gamma_{CE}$ can be constructed from the empty state. Note that the empty state trivially satisfies the preconditions of Theorem 7.1.1. The full code of the counter example is as follows:

```
1 function element(x){
2   this.next=null;
3   this
4 }
5 function break(x){
6   x.next= new element(42);
7   x.val=5;
8   x.next.val
9 }
10 break(new element(42))
```

The following sections present the formal derivation of the evaluation and typing relation.

7.1.5.1 Evaluation

The evaluation to `stuckErr` is proven in 3 parts. The first part evaluates the `element` function; the second evaluates the `break` function; finally those two function deriva-

tions are put together to form a derivation for the main function call `break(new element(42))`.

element

The expression `new element(42)` is needed twice in this example. The following is the evaluation of this expression parameterised by H, χ, ι. It uses the following modified states:

$$\begin{aligned}\chi_e &= \{\texttt{this} \mapsto \iota, \texttt{x} \mapsto 42\} \\ H_2 &= H[\iota \mapsto \ll\gg] \\ H' &= H[\iota \mapsto \ll \texttt{next} \mapsto \texttt{null} \gg]\end{aligned}$$

The function `element` is defined via

$$\mathbb{P}(\texttt{element}) = \texttt{function element(x)\{this.next=null;this\}}$$

and therefore its evaluation can be derived as:

$$\dfrac{\dfrac{\dfrac{\chi_e(\texttt{this}) = \iota}{\texttt{this}, H_2, \chi_e \twoheadrightarrow \iota, H_2, \chi_e}\,\text{(S-VAR)} \qquad \dfrac{}{\texttt{null}, H_2, \chi_e \twoheadrightarrow \texttt{null}, H_2, \chi_e}\,\text{(S-VAL)} \qquad H' = H_2[\iota \mapsto H_2(\iota)[\texttt{next} = \texttt{null}]]}{\texttt{this.next=null}, H_2, \chi_e \twoheadrightarrow \texttt{null}, H', \chi_e}\,\text{(S-MEMASS)} \qquad \dfrac{\chi_e(\texttt{this}) = \iota}{\texttt{this}, H', \chi_e \twoheadrightarrow \iota, H', \chi_e}\,\text{(S-VAL)}}{\texttt{this.next=null;this}, H_2, \chi_e \twoheadrightarrow \iota, H', \chi_e}\,\text{(S-SEQ)}$$

$$\dfrac{\dfrac{}{\texttt{42}, H, \chi \mapsto \texttt{42}, H, \chi}\,\text{(S-VAL)} \qquad \iota \text{ is new in } H \text{ and } H_2 = H[\iota \mapsto \ll\gg] \qquad \dfrac{\vdots}{\texttt{this.next=null;this}, H_2, \chi_e \twoheadrightarrow \iota, H', \chi_e}}{\texttt{new element(42)}, H, \chi \twoheadrightarrow \iota, H', \chi}\,\text{(S-NEW)}$$

break

The body of the function `break` consists of 3 statements. Each statement modifies the heap and therefore 4 different heap states are necessary.

$$\begin{aligned}H_{b0} &= \{\iota_x \mapsto \ll \texttt{next} : \texttt{null} \gg\} & \chi_{b0} &= \{\texttt{x} \mapsto \iota_x, \quad \texttt{this} \mapsto \texttt{null}\} \\ H_{b1} &= \{\iota_x \mapsto \ll \texttt{next} : \texttt{null} \gg, & & \iota_n \mapsto \ll \texttt{next} : \texttt{null} \gg\} \\ H_{b2} &= \{\iota_x \mapsto \ll \texttt{next} : \iota_n \gg, & & \iota_n \mapsto \ll \texttt{next} : \texttt{null} \gg\} \\ H_{b3} &= \{\iota_x \mapsto \ll \texttt{next} : \iota_n, \texttt{val} : 5 \gg, & & \iota_n \mapsto \ll \texttt{next} : \texttt{null} \gg\}\end{aligned}$$

With those states, the 3 statements can be evaluated separately as follows:

$$\dfrac{\dfrac{\chi_{b0}(\texttt{x}) = \iota_x}{\texttt{x}, H_{b0}, \chi_{b0} \twoheadrightarrow \iota_x, H_{b0}, \chi_{b0}}\,\text{(S-VAR)} \qquad \dfrac{\vdots}{\texttt{new element(42)}, H_{b0}, \chi_{b0} \twoheadrightarrow \iota_n, H_{b1}, \chi_{b0}} \qquad H_{b2} = H_{b1}[\iota_x \mapsto H_{b1}(\iota_x)[\texttt{next} \mapsto \iota_n]]}{\texttt{x.next=new element(42)}, h_{b0}, \chi_{b0} \twoheadrightarrow \iota_n, H_{b2}, \chi_{b0}}\,\text{(S-MEMASS)}$$

$$\dfrac{\dfrac{\chi_{b0}(\mathtt{x}) = \iota_x}{\mathtt{x}, H_{b2}, \chi_{b0} \twoheadrightarrow \iota_x, H_{b2}, \chi_{b0}}\,(\text{S-VAR}) \qquad \dfrac{}{5, H_{b2}, \chi_{b0} \twoheadrightarrow 5, H_{b2}, \chi_{b0}}\,(\text{S-VAL}) \qquad H_{b3} = H_{b2}[\iota_x \mapsto H_{b2}[\mathtt{val} \mapsto 5]]}{\mathtt{x.val{=}5}, H_{b2}, \chi_{b0} \twoheadrightarrow 5, H_{b3}, \chi_{b0}}\,(\text{S-MEMASS})$$

$$\dfrac{\dfrac{\dfrac{\chi_{b0}(\mathtt{x}) = \iota_x}{\mathtt{x}, H_{b3}, \chi_{b0} \twoheadrightarrow \iota_x, H_{b3}, \chi_{b0}}\,(\text{S-VAL}) \qquad H_{b3}(\iota_x)(\mathtt{next}) = \iota_n}{\mathtt{x.next.val}, H_{b3}, \chi_{b0} \twoheadrightarrow \mathtt{stuckErr}, H_{b3}, \chi_{b0}}\,(\text{S-MEMSEL}) \qquad H_{b3}(\iota_n)(\mathtt{val}) = \mathcal{U}df}{\mathtt{x.next.val}, H_{b3}, \chi_{b0} \twoheadrightarrow \mathtt{stuckErr}, H_{b3}, \chi_{b0}}\,(\text{S-NOMEMBER})$$

In summary, the evaluation of the whole function body of `break` evaluates in the following way:

$$\dfrac{e_1, h_{b0}, \chi_{b0} \twoheadrightarrow \iota_n, H_{b2}, \chi_{b0} \qquad \dfrac{e_2, H_{b2}, \chi_{b0} \twoheadrightarrow 5, H_{b3}, \chi_{b0} \qquad e_3, H_{b3}, \chi_{b0} \twoheadrightarrow \mathtt{stuckErr}, H_{b3}, \chi_{b0}}{\mathtt{x.val{=}5;x.next.val}, H_{b2}, \chi_{b0} \twoheadrightarrow \mathtt{stuckErr}, H_{b3}, \chi_{b0}}\,(\text{S-SEQ})}{\mathtt{x.next{=}new\ element(42);x.val{=}5;x.next.val}, H_{b0}, \chi_{b0} \twoheadrightarrow \mathtt{stuckErr}, H_{b3}, \chi_{b0}}\,(\text{S-SEQ})$$

main

In order to evaluate the main expression of this example, 3 different heaps are necessary. The stack is not changed throughout the whole evaluation:

$$H_0 = \{\} \qquad \chi_0 = \{\mathtt{x} \mapsto 2, \mathtt{this} \mapsto \mathtt{null}\}$$
$$H_1 = \{\iota_x \mapsto \ll \mathtt{next} : \mathtt{null} \gg\}$$
$$H_2 = \{\iota_x \mapsto \ll \mathtt{next} : \iota_n, \mathtt{val} : 5 \gg, \iota_n \mapsto \ll \mathtt{next} : \mathtt{null} \gg\}$$

With these states the evaluation can be derived as

$$\dfrac{\dfrac{\vdots}{\mathtt{new\ element(42)}, H_0, \chi_0 \twoheadrightarrow \iota_x, H_1, \chi_0}\,(\text{S-NEW}) \qquad \dfrac{\vdots}{e_b, H_1, \chi_0 \twoheadrightarrow \mathtt{stuckErr}, H_2, \chi_0}\,(\text{S-SEQ})}{\mathtt{break(new\ element(42))}, H_0, \chi_0 \twoheadrightarrow \mathtt{stuckErr}, H_2, \chi_0}\,(\text{S-FUNCCALL})$$
$$\mathbb{P}(\mathtt{break}) = \mathtt{function\ break(x)}\{e_b\}$$

Therefore, the evaluation results in `stuckErr`. In real world JavaScript, accessing an undefined field of an object results in the return value `undefined`. This value is not what the developer expects when accessing `x.next.val` and the system JS_0^T aims to find values which are different than expected. This is the reason, why JS_0^T correctly returns `stuckError` in this case.

7.1.5.2 Type checking

The type check for the whole example requires the following object types which coincide with the 4 steps to prepare the exploited object: creation of an empty object,

assignment to `next`, folding into a recursive type, assignment to `val`

$$O_{xe} = [\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (O_{L\circ}, \circ)]$$
$$O_{x\circ} = [\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (O_{L\circ}, \bullet)]$$
$$O_{L\circ} = \mu\alpha.[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)]$$
$$O_{L\bullet} = \mu\alpha.[\texttt{val} : (\texttt{Int}, \bullet), \texttt{next} : (\alpha, \bullet)]$$

The proof requires the property $O_{x\circ} \leq O_{L\circ}$ for which the reflexivity of the subtyping relation is useful:

$$\dfrac{\dfrac{}{t \equiv t}\ (\equiv \text{REFLEX})}{t \leq t}\ (\leq \text{CONG})$$

$O_{L\circ}$ can be expressed as $O_{L\circ} = \mu\alpha.M$ with $M = [\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)]$. Therefore, ($\equiv$ UNFOLD) is applicable in

$$\dfrac{\dfrac{\dfrac{}{O_{L\circ} \equiv O_{x\circ}}\ (\equiv \text{UNFOLD})}{O_{x\circ} \equiv O_{L\circ}}\ (\equiv \text{SYM})}{O_{x\circ} \leq O_{L\circ}}\ (\leq \text{CONG})$$

since $M[\alpha/O_{L\circ}] = [\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (O_{L\circ}, \bullet)] = O_{x\circ}$

In order to check the type of the whole program, first the types for the functions in $\mathbb{P}$ need to be validated:

$$\texttt{element}: \quad G_e = \quad O_{xe}, \texttt{Int} \rightarrow \quad O_{x\circ}$$
$$\texttt{break}: \quad G_b = \quad [], O_{L\circ} \rightarrow \quad \texttt{Int}$$

element

Two new contexts are necessary to type the body of the function `element`. The first represents the typing at the beginning of the function body where `this` is an empty object. In the second, `this` has been expanded with the field `next`.

$$\Gamma_{e0} = \{\texttt{x} \mapsto \texttt{Int}, \texttt{this} \mapsto O_{xe}\}$$
$$\Gamma_{e1} = \{\texttt{x} \mapsto \texttt{Int}, \texttt{this} \mapsto O_{x\circ}\}$$

In those contexts the body of `element` can be typed as $G_e(ret) = O_{x\circ}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{P}, \Gamma_{e0} \vdash \texttt{null} : O_{L\circ} | \Gamma_{e0}}\ (\text{T-CONST})}{\texttt{this=e}' \text{ is no subexpression of } \texttt{null} \quad \Gamma_{e0}(\texttt{this}) = O_{xe} \quad O_{xe}(\texttt{next}) = (O_{L\circ}, \circ) \quad O_{L\circ} \leq O_{L\circ} \quad \Gamma_{e1} = \Gamma_{e0}[\texttt{this} \mapsto O_{x\circ}[\texttt{next} \mapsto (O_{L\circ}, \bullet)]]}{\mathbb{P}, \Gamma_{e0} \vdash \texttt{this.next=null} : O_{L\circ} | \Gamma_{e1}}\ (\text{T-ASSIGNADD}) \quad \dfrac{\Gamma_{e1}(\texttt{this}) = O_{x\circ}}{\mathbb{P}, \Gamma_{e1} \vdash \texttt{this} : O_{x\circ} | \Gamma_{e1}}\ (\text{T-VAR})}{\mathbb{P}, \Gamma_{e0} \vdash \texttt{this.next=null;this} : O_{x\circ} | \Gamma_{e1}}\ (\text{T-SEQ})$$

This verifies the function type G_e for the function `element`.

break

Two new contexts are used to type the function body `break`. The first is the entrance context for this body and in the second the object `x` has been extended by the field `val`:

$$\Gamma_{b0} = \{\mathtt{x} \mapsto O_{L\circ}, \mathtt{this} \mapsto []\}$$
$$\Gamma_{b1} = \{\mathtt{x} \mapsto O_{L\bullet}, \mathtt{this} \mapsto []\}$$

The typing derivations for the three statements in this function body are presented separately:

$$\dfrac{\dfrac{\Gamma_{b0}(\mathtt{x}) = O_{L\circ}}{\Gamma_{b0} \vdash \mathtt{x} : O_{L\circ} | \Gamma_{b0}}\,(\text{T-VAR}) \quad O_{x\circ} \le O_{L\circ} \quad O_{L\circ}(\mathtt{next}) = (\alpha, \bullet)[O_{L\circ}/\alpha] = (O_{L\circ}, \bullet) \quad \dfrac{\dfrac{}{\Gamma_{b0} \vdash 42 : \mathtt{Int} | \Gamma_{b0}}\,(\text{T-CONST}) \quad \mathtt{Int} \le \mathtt{Int} \quad \mathbb{P}(\mathtt{element}) = \mathtt{function\ element(x)} : G_e\{\ldots\} \quad G_e(\mathtt{this}) \text{ has no } \bullet \text{ fields}}{\Gamma_{b0} \vdash \mathtt{new\ element(42)} : O_{x\circ} | \Gamma_{b0}}\,(\text{T-CALL})}{\Gamma_{b0} \vdash \mathtt{x.next=new\ element(42)} : O_{x\circ} | \Gamma_{b0}}\,(\text{T-ASSIGNEUPD})$$

$$\dfrac{\dfrac{}{\Gamma_{b0} \vdash 5 : \mathtt{Int} | \Gamma_{b0}}\,(\text{T-CONST}) \quad \mathtt{Int} \le \mathtt{Int} \quad \mathtt{x=e'} \text{ is no subexpression of } 5 \quad \Gamma_{b0}(\mathtt{x}) = O_{L\circ} \quad O_{L\circ}(\mathtt{val}) = (\mathtt{Int}, \circ) \quad \Gamma_{b1} = \Gamma_{b0}[\mathtt{x} \mapsto O_{L\circ}[\mathtt{val} \mapsto (\mathtt{Int}, \bullet)]]}{\Gamma_{b0} \vdash \mathtt{x.val=5} : \mathtt{Int} | \Gamma_{b1}}\,(\text{T-ASSIGNADD})$$

$$\dfrac{\dfrac{\dfrac{\Gamma_{b1}(\mathtt{x}) = O_{L\bullet}}{\Gamma_{b1} \vdash \mathtt{x} : O_{L\bullet} | \Gamma_{b1}}\,(\text{T-VAR}) \quad O_{L\bullet}(\mathtt{next}) = (\alpha, \bullet)[O_{L\bullet}/\alpha] = (O_{L\bullet}, \alpha)}{\Gamma_{b1} \vdash \mathtt{x.next} : O_{L\bullet} | \Gamma_{b1}}\,(\text{T-MEMACC}) \quad O_{L\bullet}(\mathtt{val}) = (\mathtt{Int}, \bullet)}{\Gamma_{b1} \vdash \mathtt{x.next.val} : \mathtt{Int} | \Gamma_{b1}}\,(\text{T-MEMACC})$$

Due to the rule (T-SEQ), the whole function body can be typed as `Int` which verifies the function type G_b for the function *break*.

Finally, the main expression `break(new element(42))` can be typed as `Int`.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 42 : \mathtt{Int} | \Gamma}\,(\text{T-CONST}) \quad \mathtt{Int} \le (\mathtt{Int} = \Gamma(\mathtt{x})) \quad G_e(\mathtt{this}) \text{ has no } \bullet \text{ fields} \quad \mathbb{P}(\mathtt{element}) = \mathtt{function\ element(x)} : G_e\{\ldots\}}{\Gamma \vdash \mathtt{new\ element(42)} : O_{x\circ} | \Gamma}\,(\text{T-CALL}) \quad O_{x\circ} \le O_{L\circ} \quad G_b(\mathtt{this}) \text{ has no } \bullet \text{ fields} \quad \mathbb{P}(\mathtt{break}) = \mathtt{function\ break(x)} : G_b\{\ldots\}}{\Gamma \vdash \mathtt{break(new\ element(42))} : \mathtt{Int} | \Gamma}\,(\text{T-CALL})$$

This shows that the code of the counter example is types as `Int`.

In summary, this section presented a full JavaScript program which can be evaluated to a `stuckError` starting in the empty state, but is typed as `Int` in JS_0^T. In contrast to the first small counter example, this is independent of the choice of $\mathcal{T}'$ and therefore shows the type checking rules of JS_0^T unsound.

7.1.6 Other weaknesses in JS_0^T

In [9] Chapter 3.2, the author remarks that multiple parameters can be modelled as fields in one parameter object. This translation is equivalent for the semantics. However, the type system allows object types to be extended with additional fields. The types of the members of object types always have to stay congruent, which requires the structure to remain unchanged throughout the execution. So, if a function is implemented with multiple parameters, each parameter has a separate object type and can be dynamically extended by assigning a value to a potential field. Using the parameter block object, the block itself can be extended but each parameter must keep its structure and can therefore not be extended. This weakness is not significant since it should not be difficult to extend the type system to handle functions with multiple parameters by other means.

7.2 Fixing soundness

The type inference algorithm in [9] is defined separately from the type checking rules. Theorem 7.2.1 then connects type checking with type inference by proving that the result of the type inference algorithm also satisfies the type checking relation:

Theorem 7.2.1 ([9, p. 96])**.** *Checking inferred types*

For program a P, expression e, pre-environment γ*, solution S, if:*

$$\begin{aligned} \gamma \vdash e : \mathbb{e} | \gamma' | C \\ S \vdash C \\ S, \mathbb{P} \vdash e \\ \mathbb{P} = \mathcal{T}(P, S) \\ \Gamma = \Gamma_{gen}(\gamma, S) \\ \Gamma' = \Gamma_{gen}(\gamma', S) \end{aligned}$$

then there exists t such that:

$$\begin{aligned} \mathbb{P}, \Gamma \vdash e : t | \Gamma' \\ t \leq S([\![\mathbb{e}]\!]) \end{aligned}$$

In order to understand the parts of this theorem, it is necessary to understand the *labels* added by the type inference. During the inference, different states of the variables

are treated with different types. Take, for example, the expression `x.m=4` which adds the field `m` to the variable `m`. In this case the type for `x` before and after the expression has to be reasoned about differently. To distinguish the different states of the variables the type inference adds unique identifiers as labels to variables. In the example, the type of `x` before the assignment `x.m=4` might be stored in the type variable $\mathtt{x}_l$. Afterwards a new label l' is used with the type variable $\mathtt{x}_{l'}$. With this in mind, the theorem speaks about the following constructs:

- The pre-environment γ is a mapping from each variable name to its current label. This identifies the type-variable to be used for this variable in the current typing environment.

- $\mathbb{e}$ is the labeled expression corresponding to e, where each occurrence of each variables has been labeled.

- The solution S maps labeled type-variables to full types.

- The function $\Gamma_{gen}(\gamma, S)$ uses the solution S to complete the pre-environment γ into an environment $\{\mathtt{this} \mapsto S(this\_\gamma(this)), \mathtt{x} \mapsto S(x\_\gamma(x))\}$.

- The function $\mathcal{T}(P,S)$ adds the function-types to the function definitions in the program P according to the solution S to obtain the types program $\mathbb{P}$.

The preconditions (see [9] for formal definitions) intuitively assert the following:

- $\gamma \vdash e : \mathbb{e}|\gamma'|C$: In the pre-environment the expression e can be labeled as $\mathbb{e}$ and produce the new pre-environment γ' and the constraints C during execution.

- $S \vdash C$: The solution S solves all the constraints in C.

- $S, \mathbb{P} \vdash e$: All functions occurring in e are defined in $\mathbb{P}$ and typed consistent with the solution S.

The main claim of the theorem is that the expression e can be type checked with a subtype of the result of the type inference algorithm.

7.2.1 Relation between type checking and type inference

The proof for this theorem contains a similar fault as in the type checking soundness theorem: Consider again the expression `x.val=3` in the solution

$$\begin{aligned} S: \quad & x\_1 \mapsto \mu\alpha.[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)] \\ & x\_2 \mapsto [\texttt{val} : (\texttt{Int}, \bullet), \texttt{next} : (\mu\alpha.[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)], \bullet)] \end{aligned}$$

The labels $1, 2$ here are the labels for the variable `x` before and after the evaluation of `x.val=3`. The type for `x_1` has a state $\circ$ for the field `val` before the expression. After the execution the state in the new type x_2 has changed to $\bullet$ and the field `next` has been unfolded.

For this state the proof derives the facts:

$$S([\![x\_1]\!])(\texttt{val}) = (t'', \psi) \qquad \text{with } t'' = \texttt{Int} \tag{73}$$
$$S([\![x\_2]\!])(\texttt{val}) = (t''', \bullet) \qquad \text{with } t''' = \texttt{Int} \tag{75}$$
$$\forall m' \neq \texttt{val} : S([\![x\_2]\!])(m') \equiv S([\![x\_1]\!])(m') \tag{77}$$
$$S([\![x\_2]\!])(\texttt{val}) \leq S([\![x\_1]\!])(\texttt{val}) \tag{78}$$
$$t'' \equiv t''' \tag{80}$$

In the considered example, all these facts are true:

- (77): The only other candidate for m' is $m' = \texttt{next}$ and

$$\begin{aligned} S([\![x\_1]\!])(\texttt{next}) &= \alpha[\alpha / S([\![x\_1]\!])] \\ &= \mu\alpha.[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)] \\ &= S([\![x\_2]\!])(\texttt{next}) \end{aligned}$$

- (78): $S([\![x\_2]\!])(\texttt{val}) = (\texttt{Int}, \bullet) \leq (\texttt{Int}, \circ) = S([\![x\_1]\!])(m)$ with $m = \texttt{val}$
- (80): $t'' = t''' = \texttt{Int}$ and therefore trivially $t'' \equiv t'''$

Line (91) then states:

From (77),(78),(73) and (80) we get:

$$S([\![var\_l']\!]) = S([\![var\_l]\!])[m \mapsto (t'', \bullet)] \tag{91}$$

which applied to the considered example amounts to

$$S([\![x\_2]\!]) = S([\![x\_1]\!])[\texttt{val} \mapsto (\texttt{Int}, \bullet)]$$

with the following values obtained from the solution S

$$S([\![x\_2]\!]) = [\texttt{val} : (\texttt{Int}, \bullet), \texttt{next} : (\mu\alpha.[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)], \bullet)]$$
$$S([\![x\_1]\!])[\texttt{val} \mapsto (\texttt{Int}, \bullet)] = \mu\alpha.[\texttt{val} : (\texttt{Int}, \bullet), \texttt{next} : (\alpha, \bullet)]$$

The two types are obviously not equal. In the proof congruence would suffice, but since the types of the field `next` are not congruent (see Proposition 7.1.4), congruence does not hold, either:

$$S([\![var\_2]\!])(\texttt{next}) = \mu\alpha.[\texttt{val} : (\texttt{Int}, \circ), \texttt{next} : (\alpha, \bullet)]$$
$$\begin{aligned} S([\![var\_1]\!])[\texttt{val} \mapsto (\texttt{Int}, \bullet)](\texttt{next}) &= \alpha[\alpha / S([\![var\_1]\!])[\texttt{val} \mapsto (\texttt{Int}, \bullet)]] \\ &= \mu\alpha.[\texttt{val} : (\texttt{Int}, \bullet), \texttt{next} : (\alpha, \bullet)] \end{aligned}$$

As a result, the proof of correspondence between type inference and type checking is broken as well. Fortunately, those two faults seem to cancel each other out and therefore the broken rule (T-ASSIGNADD) can be fixed imitating the inference rule (CG-ASSIGNADD).

7.2.2 Corrected inference rule

The corresponding type inference rule to the type checking rule (T-ASSIGNADD) is (CG-ASSIGNADD)

$$\frac{\begin{array}{c} \gamma \vdash e : \mathbb{e} | \Gamma'' | C' \\ \gamma''(var) = l \\ l' \notin \gamma''(lab) \\ \gamma' = \gamma''[var \mapsto l', lab \mapsto (\gamma''(lab) \cup \{l'\})] \\ var = e' \text{ is not a subexpression of } e_2 \\ C = \{[\![var\_l]\!] \leq [m : ([\![var\_l.m]\!], \circ)], \\ [\![var\_l']\!] \leq [m : ([\![var\_l'.m]\!], \bullet)], \\ [\![var\_l']\!] \lhd_m [\![var\_l]\!], \\ [\![\mathbb{e}]\!] \leq [\![var\_l'.m]\!], \\ [\![\mathbb{e}]\!] \leq [\![var\_l'.m = \mathbb{e}]\!]\} \end{array}}{\gamma \vdash var.m = e : var\_l'.m = \mathbb{e} | \gamma' | C \cup C'} \quad \text{(CG-ASSIGNADD)}$$

The main difference to (T-ASSIGNADD) is that the rule (CG-ASSIGNADD) uses the constraint $[\![var\_l']\!] \lhd_m [\![var\_l]\!]$ to specify the fields of the extended object. In contrast, (T-ASSIGNADD) declares the new type directly using $\Gamma'' = \Gamma'[\mathtt{var} \mapsto O[m \mapsto (t'', \bullet)]]$ and therefore simply copies all remaining fields $m' \neq m$ which causes the erroneous result. The constraint approach can be directly ported to the type checking rule. Define the constraint $\lhd_m$ as abbreviation of the following set of subtyping and congruence relations:

$$\frac{\forall m' \neq m : O'(m') \equiv O(m') \qquad O'(m) \leq O(m)}{O' \lhd_m O} \qquad (\lhd_m \text{OBJ})$$

With this definition, the broken rule (T-ASSIGNADD) can be modified to

$$\frac{\begin{array}{c}\mathbb{P}, \Gamma \vdash e_2 : t | \Gamma' \\ \mathtt{var}{=}e' \text{is not a subexpression of } e_2 \\ \Gamma'(\mathtt{var}) = O \\ O(m) = (t'', \psi) \\ t \leq t'' \\ O' \lhd_m O \\ \Gamma'' = \Gamma'[\mathtt{var} \mapsto O']\end{array}}{\mathbb{P}, \Gamma \vdash \mathtt{var}.m = e_2 : t | \Gamma''} \qquad \text{(T-ASSIGNADD*)}$$

The proof of soundness can be corrected in a similar way. Instead of constructing the new heap typing $\mathcal{T}'$ by assignment, it can be constructed as

$$\mathcal{T}' = \mathcal{T}''[\iota \mapsto O'] \text{ with } O' \lhd_m O$$

Such a type O' can always be constructed: if the type O is recursive and therefore of the form $\mu\alpha.M$, then unfold one step to make the type not recursive on the outer most level $M[\alpha/\mu\alpha.M]$. After that the state of m can be modified to $\bullet$ if necessary since modifying the fields of this new object does not interfere with the deeper structure of the object.

For this new construction of $\mathcal{T}'$, the fact $\mathcal{T}' \sqsubseteq T''$ in the soundness proof (line (27), p.66) now holds:

Lemma 7.2.2 ([9, (27)p. 66])**.** *Extension soundness*
Given

$$\mathcal{T}''[\mathfrak{l}] = O$$
$$\mathcal{T}' = \mathcal{T}''[\mathfrak{l} \mapsto O'] \text{ with } O' \lhd_m O$$

it holds that

$$\mathcal{T}' \sqsubseteq \mathcal{T}''.$$

Proof. By the definition of $\mathcal{T}'$ it is clear that $\mathfrak{l} \in \mathtt{Dom}(\mathcal{T}')$. The precondition ensures $\mathfrak{l} \in \mathtt{Dom}(\mathcal{T}'')$. All other addresses are not changed in $\mathcal{T}'$ in comparison to $\mathcal{T}''$ and therefore $\mathtt{Dom}(\mathcal{T}') = \mathtt{Dom}(\mathcal{T}'')$. Since only $\mathfrak{l}$ has been changed, all that remains to be shown is that $\mathcal{T}'(\mathfrak{l}) \sqsubseteq \mathcal{T}''(\mathfrak{l})$, which reduces to $O' \sqsubseteq O$.

By Definition of $\lhd_m$ it holds

$$O'(m) \leq O(m)$$
$$\forall m' \neq m. O'(m') \equiv O(m')$$

Since the 3 relations $\mathcal{U}df \equiv t$, $\mathcal{U}df \leq t$ and $t \leq \mathcal{U}df$ are only true for $t = \mathcal{U}df$, it is obvious that $O(m) = \mathcal{U}df \Leftrightarrow O'(m) = \mathcal{U}df$. For m $O'(m) \leq O(m)$ is given by $O' \lhd_m O$. For every other $m' \neq m$, the relation $O'(m') \equiv O(m')$ holds due to $O' \lhd_m O$. That induces $O'(m') \leq O(m')$ by the rule ($\leq$ CONG). Together this proves $O'(m'') \leq O(m'')$ for all $m'' \in \mathtt{Dom}(O)$ and therefore $O' \sqsubseteq O$, which concludes the proof. □

Chapter 8

Conclusion

8.1 Summery

In this thesis I documented the results of my research into quantitative bounds on the resource consumption of apps written in JavaScript. The presented research imposes bounds restricting the app to a finite number of resource accesses instead of granting unrestricted access. To capture the resource behaviour of modern apps, the bounds depend on the size of data structures and on the way and extend the user interacts with the user interface of the app. With these properties a policy can differentiate legitimate functionality from unreasonable resource usage. The current access control for resources on mobile phones based on permissions is not fine-grained enough to make this distinction.

As a side-effect, the interaction dependent bounds intuitively describe the functionality of the app. For example, with a button policy allowing one send message per click on the button “Send” a user can assume the app has a messaging feature. Policies for multiple resources can quickly establish a comprehensive image of the feature set of the app. This interaction-centric description of the functionality of an app can amend or replace the free form description provided by the developer in the Google Play Store.

Based on previous research, I have developed two different systems to produce such bounds with different properties.

The dynamic system PhoneWrap presented in Chapter 4 enforces given bounds during runtime. This is achieved by inserting a wrapping script at the top of the app’s HTML file, which overwrites all resource consuming APIs with an instrumented version. The instrumented APIs first check the given policy and, if the allowed resource

access is exceeded, a deny behaviour is executed instead of the requested API. The bounds enforced by PhoneWrap are described in terms of tickets, which each grant one-time access to the resource. The policy describes how many tickets the functionality triggered by each UI element requires and generates exactly the specified amount for each user interaction with each link, button, checkbox or dialog. This way, the enforced bounds on the resource consumption are interaction and context dependent. PhoneWrap can insert given policies automatically into PhoneGap apps for Android downloaded from the Google Play store. The injected app can be executed on an unmodified Android device and is limited by the resource bound specified in the policy. PhoneWrap can be used by experienced device users to insert individualised resource bounds. Alternatively, third party policy authors can publish the modified app package or the policy file such that users can use PhoneWrap and insert the wanted policy themselves.

Via the runtime enforcement PhoneWrap only considers the resource usage of the executed path and the actually consumed resources. The policy can alter the behaviour of the app, if the bound is exceeded. This way it can restrict potentially malicious apps to execute the wanted functionality without allowing access to the resource for unwanted part of the app. A malicious app might still abuse the granted tickets. However, in that case the missing functionality is an indicator for the abuse and the damage is limited by the tickets. Another advantage of the runtime monitoring is that dynamic features of JavaScript, like the `eval` function are guarded by the policy as any other code. The system has been successfully verified on a set of real world Android apps and was able to correct the resource behaviour in one case.

The injection scripts of PhoneWrap are Android specific, but the written policies are platform independent for all PhoneGap apps and the wrapper script enforces a given policy in any JavaScript environment.

The system AmorJiSe statically derives bounds on the resource behaviour of the given JavaScript code. The types and typing rules of AmorJiSe are formally specified in Chapter 6 and the correctness of the resulting bounds is proven mathematically in relation to the formalised evaluation rules. Thus, the resulting bounds hold without the need to trust a third party implementation. As static type system AmorJiSe is able to analyse programs before execution and over-approximates all possible runtime executions of the original code. This changes neither the behaviour nor the performance of the code. During analysis the inference of the resource annotations is reduced to a Linear Programming Problem, which can be solved by conventional solvers in polyno-

mial time. Given the full resulting types, the correspondence to the analysed code and the correctness of the annotations can be checked without solving the full LPP. Therefore, the types qualify as digital evidence. They can be generated by a third party and checked even on a smaller device like a phone without the need to trust the evidence creator .

The types of AmorJiSe build upon existing data types which describe the data structure of the values. Depending on the choice of this underlying system the type inference can have different properties. However, the annotations added to the data types are in any case inferred without the need to read or edit the analysed code. I have discussed different candidates for the underlying type system in Chapter 5 and used JS_0^T for an implementation. While working with the system JS_0^T, I identified faults in the soundness proof of its type checking and in the connection between type checking and type inference. In Chapter 7 I discussed a counter examples in details and proposed a solution fixing both errors. With the corrected rules, I have proven the correctness of the faulty proof steps in the original proof.

The core of JavaScript AmorJiSe analyses is platform independent. Therefore, AmorJiSe is not restricted to Android or mobile apps in general.

The two systems PhoneWrap and AmorJiSe provide similar bounds. The reserved resource units within AmorJiSe types are in many ways equivalent to the tickets granted in PhoneWrap policies. The API policies specified for PhoneWrap can be directly translated into API types in the initial typing context for AmorJiSe. The tickets granted initially in PhoneWrap correspond to the constant resource usage in the typing judgement of AmorJiSe and the bounds specified in PhoneWrap's interaction dependent policies can be extracted from the function types of the event handlers in the typing context resulting from AmorJiSe.

Due to their different properties, each system is preferable in different situation. AmorJiSe can be used before installation to verify the absence of malicious behaviour or compare the resource requirements of different apps. PhoneWrap, on the other hand, can be used to suppress potentially included malicious behaviour or disable parts of the functionality of the app.

8.2 Further work

There is a number of ways in which the presented systems could be extended.

The dependence of the bounds on user interaction events is motivated by use cases.

However, there is nothing special about user event and the systems could easily be extended to additional events like `lowbattery`, `online` or time. This leads to a new set of possible policies. With PhoneWrap an app could be restricted not to use the data-connection when the battery is low, only to send messages when Internet messaging is not available or to access the location at most once per hour.

Another area of research would be the combination of the two systems. The results of the two different systems could be used to enhance each other in different ways. First of all the individual results can be used to reduce the disadvantages of the other system: The bounds inferred by AmorJiSe are mathematically guaranteed. If AmorJiSe is able to proof a bound requested in the PhoneWrap policy the enforcement code of PhoneWrap can be omitted and the runtime overhead reduced. The other way round, AmorJiSe should be able to type an app injected with the PhoneWrap system to produce a mathematical proof of a bound reasonably close to the requested bound.

Ideally the two methods should be integrated, such that AmorJiSe analyses an app with a target bound. If this bound cannot be achieved statically in certain places, for example, for the `eval` function, it inserts dynamic enforcement code and type checks the enforced code. With the partial static information about the resource consumption, the dynamically enforced bounds can be chosen to suit the bound inference. This hybrid resource analysis would provide mathematically proven bounds with runtime enforcement only where necessary.

Appendix A

PhoneWrap code

```
1  (function() {
2    var original = {};
3    var policy;
4    var handlers = null; // list of handlers attached to deviceready
5    var expecting_deviceready = true;
6
7
8    // initialize policy --------------------
9    policy = {}
10   //---------------------------------------
11
12 //    var policy_creation = true;
13   //Default values
14   if (typeof policy.blockAll === "undefined") policy.blockAll =      false;
         // if true no accesses are granted no matter what the tickets say.
15   if (typeof policy.allowAll === "undefined") policy.allowAll =      false;   //
          if true all accesses are granted no matter what the tickets say (unless
         blockAll)
16   if (typeof policy.generate === "undefined") policy.generate =      true;    //
          used internally: if false does not generate any new tickets
17   if (!policy.mperms)                         policy.mperms =        0;
18   if (!policy.buttons)                        policy.buttons =       [];
19   if (!policy.deny)                           policy.deny =          function()
         {};
20   if (!policy.guard)                          policy.guard =         [];
21   if (!policy.guard_exec)                     policy.guard_exec =    [];
22   if (!policy.guard_require)                  policy.guard_require = [];
23
24   policy.mperms_local = 0;
25   policy.mperms_global = policy.mperms;
26   delete policy.mperms;
27
28   var deviceReadyFired = false; //log if device ready was fired and don't fire
         it again.
29
30
31   original.addEventListener = document.addEventListener;
```

```
32    original.dispatchEvent    = document.dispatchEvent;
33
34    //ToBE stored
35    // hasOwnProperty, typeof?, alert, apply?, call?, length, split, indexof,
          hasAttribute, getAttribute, match, toString?, toNummeric
36
37    var exec_guarded = function(f,f_this,f_arguments) {
38  var allowed_global = (policy.mperms_global > 0);
39  var allowed_local  = (policy.mperms_local  > 0);
40  var perms_pre;
41  if (!policy.blockAll) {
42      if (policy.allowAll) {
43    var back = f.apply(f_this,f_arguments);
44    //alert("original function executed");
45    return back;
46      }
47      if (allowed_local) {
48    perms_pre = policy.mperms_local;
49    var back = f.apply(f_this,f_arguments);
50    policy.mperms_local = perms_pre -1; //only ever decrease by 1, even if wrapped
           by this policy multiple times
51    return back;
52      }
53      if (allowed_global) {
54    perms_pre = policy.mperms_global;
55    var back = f.apply(f_this,f_arguments);
56    policy.mperms_global = perms_pre -1; //only ever decrease by 1, even if
          wrapped by this policy multiple times
57    return back;
58      }
59  }
60  return policy.deny(f_this,f_arguments);
61    }
62
63    var guard_API=function(orig) {
64  return function() {
65      return exec_guarded(orig,this,arguments);
66  }
67    }
68    var guard_exec = function(clas,meth) {
69  var ce;
70  if ((typeof Cordova !== 'undefined') && ('exec' in Cordova) ) {
71      ce = Cordova.exec;
72  }
73  if ((typeof cordova !== 'undefined') && ('exec' in cordova) ) {
74      ce = cordova.exec;
75  }
76  if ((typeof PhoneGap !== 'undefined') && ('exec' in PhoneGap) ) {
77      ce = PhoneGap.exec;
78  }
79  if (typeof ce === 'undefined') {
80      //alert('exec not found');
81      return;
```

```
82  }
83  //alert ("wrapping "  + clas + ", " + meth);
84
85  wrapped_exec = function() {
86      if ( ( arguments[2] == clas) && (arguments[3] == meth) ) {
87    return exec_guarded(ce,this,arguments);
88      } else {
89    return ce.apply(this,arguments);
90      }
91  };
92  if (typeof Cordova !== 'undefined') Cordova.exec = wrapped_exec;
93  if (typeof cordova !== 'undefined') cordova.exec = wrapped_exec;
94  if (typeof PhoneGap !== 'undefined') PhoneGap.exec = wrapped_exec;
95    };
96
97    var guard_require = function(plugin,meth) {
98  if (typeof cordova === 'undefined') return;
99  if (typeof cordova.require === 'undefined') return;
100
101 var orig_req = cordova.require;
102
103 cordova.require = function() {
104     var api_obj = orig_req.apply(this,arguments);
105     if (arguments[0] === plugin) {
106   if (typeof api_obj[meth] === 'undefined') {
107       alert(meth + " not found in plugin " + plugin);
108   } else {
109       //find exact location:
110       var iter = api_obj;
111       while (!iter.hasOwnProperty(meth)) iter = iter.__proto__
112       iter[meth] = guard_API(iter[meth]);
113   }
114     }
115     return api_obj;
116 }
117   }
118
119   var wrap_APIs = function() {
120 for (var i=0;i<policy.guard.length;i++) {
121     var iter = window;
122     var last = window;
123     var path = policy.guard[i].split(".");
124     var found = true
125     for (var j=0;j<path.length;j++) {
126   if (path[j] in iter) {
127       last = iter;
128       iter = iter[path[j]];
129   } else {
130       found = false;
131       break;
132   }
133     }
134     if (found) {
```

```
135     //alert("wrapping " + path);
136     last[path[j-1]] = guard_API(iter);
137       } else {
138     //alert('not found ' + path);
139       }
140   }
141   if (typeof navigator !== 'undefined') {
142       if (typeof navigator.notification !== 'undefined') {
143     if (typeof navigator.notification.confirm !== 'undefined') {
144 //        alert("wrapping dialog")
145         navigator.notification.confirm = guard_dialog(navigator.notification.
                confirm);
146     } //else alert("confirm not found");
147       } //else alert("notification not found");
148   } //else alert("navigator not found")
149     }
150     var wrap_exec = function() {
151   for (var i=0;i<policy.guard_exec.length;i++) {
152       var cm = policy.guard_exec[i].split(".");
153       guard_exec(cm[0],cm[1]);
154   }
155     }
156
157     var wrap_require = function() {
158 //  alert("Wrapping require apis");
159   for (var i=0;i<policy.guard_require.length;i++) {
160       var cm = policy.guard_require[i].split(".");
161       guard_require(cm[0],cm[1]);
162   }
163     }
164
165     wrap_exec();
166     wrap_APIs();
167     wrap_require();
168
169     original.addEventListener.call(document,
170             'deviceready',
171             function() {
172 //                alert("deviceready received");
173                 deviceReadyFired = true;
174                 //Do the wrapping: ----------------------------------------
175                 wrap_APIs();
176                 wrap_exec();
177                 wrap_require();
178                 // ---------------------------------------------------------
179
180                 expecting_deviceready = false;
181                 // call hold back handlers
182                 var h = handlers;
183                 while (h!== null) {
184               h.args[1].apply(h.this,arguments);
185               h=h.next;
186                 }
```

```
187                 //call hold back handlers by the cordova 'addEventListener'
188                 h = document.addEventListener.getHandlers();
189                 while (h !== null) {
190               try {
191                   h.args[1].apply(h.this,arguments);
192               }
193               finally {
194                   h=h.next;
195               }
196                 }
197             }, true
198           );
199
200     var policy_mark;
201     // listen whether additional libraries are inserted
202     var observer=new MutationObserver(function (mutations) {
203   for (var i=0;i<mutations.length;i++) {
204       if (!mutations.hasOwnProperty(i)) continue;
205       var mutation = mutations[i];
206       if (mutation.type === "childList") {
207     if (!mutation.addedNodes) continue;
208     for (var j=0;j<mutation.addedNodes.length;j++) {
209         if (!mutation.addedNodes.hasOwnProperty(j)) continue;
210         var node = mutation.addedNodes[j];
211         if (typeof node.tagName === "undefined") continue;
212         if (node.tagName.toLowerCase() == "script") {
213       //alert("script added: " + node.getAttribute("src"));
214       wrap_APIs();
215       wrap_exec();
216       wrap_require();
217       //duplicate deviceready event to make sure it arives even in old versions of
              Phonegap
218       document.addEventListener("deviceready",function() {if (!deviceReadyFired){
              deviceReadyFired = true; document.dispatchEvent(new Event('deviceready')
              );}});
219         } else if (policy_mark) {
220       var pol = eval_policy(node);
221       if ( policy_active(pol) ){
222           node.style.borderStyle = "solid";
223           node.style.borderColor = "red";
224           node.style.borderWidth = "5px";
225       }
226         }
227     }
228       } else if (mutation.type === "attributes") {
229     var node = mutation.target;
230
231     var pol = eval_policy(node);
232     if (policy_active(pol)) {
233         node.style.borderStyle = "solid";
234         node.style.borderColor = "red";
235         node.style.borderWidth = "5px";
236     }
```

```
237       }
238   }
239     });
240     var config = {subtree: true, childList: true};
241     if (policy_mark) config.attributes=true;
242     observer.observe(document,config);
243
244     //Wrap confirmation dialogs
245     var guard_dialog=function(orig) {
246   return function() {
247       //save button lables
248       var buttonlables = [ "OK","Cancel"];
249       if (arguments.hasOwnProperty(3)) {
250     buttonlables = arguments[3];
251     if (typeof buttonlables == 'string') {
252         buttonlables = buttonlables.split(','); //old version took buttons as
                comma sep list
253     }
254       }
255
256       var orig_callback = arguments[1];
257       var new_callback = function(buttonIndex) {
258     //Go through all the possible confirmable tickets
259     var buttonText = buttonlables[buttonIndex-1];
260 //    alert(buttonText);
261     for (var i=0; i<policy.confirmable.length; i++) {
262         if (!policy.confirmable.hasOwnProperty(i)) continue;
263         var cable = policy.confirmable[i];
264         if (cable.buttons.indexOf(buttonText) > -1) {
265       if (cable.local) {
266           policy.mperms_local += cable.mperms;
267           alert("Policy granted locally: " + policy.mperms_local);
268       } else {
269           policy.mperms_global += cable.mperms;
270           alert("Policy granted globally: " + policy.mperms_global);
271       }
272         }
273     }
274     //Delete all confirmables (They had their chance)
275     policy.confirmable = [];
276     //Execute original Callback
277     orig_callback.apply(this,arguments);
278       }
279
280       //Execute original dialog
281       arguments[1] = new_callback;
282       return orig.apply(this,arguments);
283   }
284     }
285
286
287
288     //Wrap inputs ------------------------------------------------
```

```
289     var iter = document;
290     while (iter.__proto__.addEventListener) {iter = iter.__proto__;}
291
292     //wrap dispatchEvent
293     iter.dispatchEvent = function() {
294   var ret;
295   var is_click = (arguments[0].type === "click");
296   if (is_click) {
297       var old = policy.generate;
298       policy.generate = false;
299   }
300   ret = original.dispatchEvent.apply(this,arguments)
301   if (is_click) {
302       policy.generate = old;
303   }
304     };
305
306     /* expected return: Object containing
307        - mperms_local : the number of local  tickets to be granted
308        - mperms_global: the number of global tickets to be granted
309        - confirm: list of confirmable tickets. Each element in the list is an
               object with the following properties:
310            - buttons: list of captions of buttons, that confirm this ticket
311      - mperms:  number of tickets that can be confirmed
312      - local:   BOOLEAN specifying whether these tickets will be local or global
313        - (optional) allowAll: new value for the allowAll parameter of the policy
314        - (optional) blockAll: new value for the blockAll parameter of the policy
315      */
316     var eval_policy = function(target) {
317   var pol = {mperms_local:0,mperms_global:0,confirm:[]};
318   for (var i=0;i<policy.buttons.length;i++) {
319       var applies = true;
320       var button = policy.buttons[i];
321       if (typeof button.cond === "undefined") button.cond = [];
322       for (var crit in button.cond) {
323     if (!target.hasAttribute(crit)) {
324         applies = false;
325         break;
326     }
327     var tcrit = target.getAttribute(crit);
328     var pcrit = button.cond[crit];
329     if (typeof button.match !== "string")
330         button.match = "exact";
331     switch (button.match) {
332     case "exact":
333           if (tcrit !== pcrit)
334       applies = false;
335         break;
336     case "different":
337         if (tcrit === pcrit)
338       applies = false
339         break;
340     case "contains":
```

```
341         if (tcrit.indexOf(pcrit) === -1 )
342       applies = false;
343         break;
344     case "regex":
345         if (! tcrit.match(pcrit) )
346       applies = false;
347         break;
348     case "begins":
349         if (tcrit.indexOf(pcrit) !== 0 )
350       applies = false;
351         break;
352     case "ends":
353         if (tcrit.indexOf(pcrit, tcrit.length - pcrit.length) === -1)
354       applies = false;
355         break;
356     default:
357         applies = false;
358     }
359       }
360
361       if (applies) {
362     var b_mperms = button.mperms;
363     // update allowAll and blockAll if neccessary:
364     if (typeof button.allowAll !== 'undefined') {
365         if (typeof pol.allowAll === 'undefined') {
366       pol.allowAll = button.allowAll;
367         } else {
368       pol.allowAll = pol.allowAll && button.allowAll;
369         }
370     }
371     if (typeof button.blockAll !== 'undefined') {
372         if (typeof pol.blockAll === 'undefined') {
373       pol.blockAll = button.blockAll;
374         } else {
375       pol.blockAll = pol.blockAll || button.blockAll;
376         }
377     }
378     // correct b_mperms depending on type of target
379     if (button.checkbox) { //defined and trueish
380         if (typeof target.checked === 'undefined') b_mperms = 0; // target is not
                what the policy asked for
381         else if (!target.checked) b_mperms = -b_mperms;               // target
                has been unchecked -> retract tickets
382     }
383     // add tickets to the correct pile in pol
384     if (button.hasOwnProperty("confirm")) {
385         //If the mperms need to be confirmed, save the condition and number
386         var cable = {buttons:button.confirm,mperms:b_mperms}; // new confirmable
387         if (typeof button.local !== "undefined") cable.local = button.local;
388         pol.confirm.push(cable);
389     } else {
390         if (!policy.generate) continue;
391         if (button.local) //defined and trueish
```

```
392       pol.mperms_local += b_mperms;
393         else
394       pol.mperms_global += b_mperms;
395     }
396       }
397   }
398   return pol;
399     };
400
401     var policy_active= function(pol) {
402   if (pol.mperms_local !== 0) return true;
403   if (pol.mperms_global !== 0) return true;
404   if (typeof pol.allowAll !== "undefined") return true;
405   if (typeof pol.blockAll !== "undefined") return true;
406   if (typeof pol.confirm !== "undefined")
407       if (pol.confirm.length > 0) return true;
408   return false;
409     }
410
411     var policy_creation;
412     var exec_policy= function(target) {
413   if (policy_creation) alert("pressed \n" + htmlElement2string(target));
414
415   var pol = eval_policy(target);
416   var granted = false;
417
418   policy.mperms_global += pol.mperms_global;
419   policy.mperms_local  += pol.mperms_local;
420   if ( (pol.mperms_local !== 0) || (pol.mperms_global !== 0) ) alert("Policy
          granted: total (" + policy.mperms_global + "," + policy.mperms_local + ")");
421   window.setTimeout(function() {policy.mperms_local = 0},0);
422
423   if (typeof pol.allowAll !== "undefined") {
424       policy.allowAll = pol.allowAll;
425       alert("Policy activated: allowAll =" + policy.allowAll);
426   }
427
428   //store confirmable tickets
429   if (pol.confirm.length > 0) {
430       alert("Policy awaits confirmation");
431       policy.confirmable = pol.confirm;
432   }
433     }
434
435     //listen for all click events at the root of DOM
436     original.addEventListener.call(document,'click',function(event){exec_policy(
          event.target)},true);
437
438     //Auxiliary functions for policy creation
439     function htmlElement2string(h) {
440   var attrs = h.attributes;
441   if (attrs.length == 0) return "";
442   var str = attrs[0].nodeName + ":" + attrs[0].nodeValue;
```

```
443   for (var i = 1;i<attrs.length;i++) {
444       if (!attrs.hasOwnProperty(i)) continue;
445       str = str + "\n" + attrs[i].nodeName + ":" + attrs[i].nodeValue;
446   }
447   return str;
448     }
449 })();
```

Appendix B

PhoneWrap policy specification

- `mperms`: Specifies the number of tickets the application is granted at launch. If this value is not specified in the policy, the default value is 0.

- `allowAll`: a Boolean specifying whether to allow access to the resource independent of the remaining tickets. If this parameter is set to true, no tickets are consumed and access is granted to all requests. If it is false, the normal ticket policy is in place. Since it can be set on button press, this can specify default policies as specified on page 52. The default value of this parameter is false.

- `blockAll`: a Boolean specifying whether to block access to the resource independent of the remaining tickets. If this parameter is set to true, all access is blocked and no tickets are consumed. If it is false, the normal ticket policy is in place. If both allowAll and blockAll are set, all access is blocked as this is the fail-safe behaviour. The default value of this parameter is false.

- `buttons`: Contains the specification for the interaction dependent policy. The value of this field is a list (JSON array) of button policies. Each button specification is again an object with the following fields. If no button policies are given, the empty list is assumed for `buttons`, which results in an interaction independent policy.

  - `cond`: a list of key-value pairs specified as a JSON object. A DOM element, which owns all of the specified attributes with the matching values will be effected by this button policy and generate the specified number of tickets for every click event. The default for this parameter is the empty set of conditions, which means every element will generate tickets on click.

- `mperms`: the number of tickets granted for each click event on the applicable DOM element
- `match`: a string that specifies how the values stored in `cond` are compared with the value of the DOM elements. Possible values are:
  - "exact": The two values need to be equal.
  - "different": Tickets are generated if the element's value does not equal the value specified in the policy.
  - "contains": Tickets are generated if the element's value contains the value specified in the policy as substring.
  - "begins": Tickets are generated if the element's value starts with the policy's value.
  - "ends": Tickets are generated if the element's value ends with the value specified in the policy.
  - "regex": The string specified in the policy is interpreted as a JavaScript regular expression and tickets are generated if the element's value matches this regular expression.

  The default value for `match` is "exact".
- `checkbox`: Boolean which specifies whether this DOM element is to be treated as a checkbox. Checkboxes generate tickets, when they get checked, but deduct the same amount of tickets, when they get unchecked.
- `confirm`: contains a list of strings. The tickets generated by this button policy are only granted, if the user afterwards confirms the action with a click on a button with one of the specified strings as caption in a confirmation dialog. If this parameter is not specified tickets are granted without confirmation.
- `allowAll`: a Boolean which specifies if after the click on the applicable buttons all accesses to the resource should be allowed. If set, the value of this parameter is copied into the allowAll parameter of the policy after this button has been pressed.
- `blockAll`: a Boolean which specifies if after the click on the applicable buttons all accesses to the resource should be blocked. If set, the value of this parameter is copied into the blockAll parameter of the policy after this button has been pressed.

  - `local`: a Boolean which specifies the expiration of the tickets generated by applicable buttons. If set to true, the tickets are deleted after all handlers of this button have been executed. If set to false, the tickets generated by this button can be used at any point after the button press.

- `guard`: a list of APIs to be included in this policy. Each call to the specified functions will be reflected by a decrease of the counter. Each API function to be guarded is specified as string. The usual JavaScript "." notation to access object fields is used. If this parameter is omitted, no APIs will be effected by this policy. More details on this follow in Section 4.3.1.

- `guard_exec`: a list of calls to `exec` that will be governed by this policy. Each call to `exec` with these parameters will be reflected in the policy state by a decrease of the ticket counter. Each element in the list is represented as string and specifies the governed class and method separated by ".". For example the parameter "SMSPlugin.send" specifies to guard the method `send` of the Java class `SMSPlugin`. If this parameter is omitted, calls to `exec` are not filtered by this policy.

- `guard_require`: a list of calls to the `require` function, which should be guarded by this policy. Each element in the list is represented as string and specifies the plugin and method of the plugin separated by ".". For example the parameter "cordova/SMSPlugin.send" instructs PhoneWrap to guard the `send` method of the plugin `cordova/SMSPlugin`. If this parameter is omitted, no calls to the `require` function will be filtered by this policy.

- `deny`: a function to be executed if a resource request is denied. The default behaviour is to ignore the function call to the guarded function without further effects. The policy object is accessible inside the deny function body. Therefore, the behaviour can change the policy by for example setting `blockAll` to `true` to deny all future requests.

Notations

Bibliography

[1] Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business Media, 1996.

[2] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 1–10. ACM, 2012.

[3] Alexander Aiken and Edward L. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41. ACM, 1993.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, pages 161–203, February 2011.

[5] Elvira Albert, Samir Genaim, and Abu Naser Masud. More precise yet widely applicable cost analysis. In *Verification, Model Checking, and Abstract Interpretation*, pages 38–53, 2011.

[6] Chaitrali Amrutkar and Patrick Traynor. Short Paper: Rethinking Permissions for Mobile Web Apps: Barriers and the Road Ahead. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 15–20, 2012.

[7] Christopher Anderson and Paola Giannini. Type Checking for JavaScript. In *Proceedings of the Second Workshop in Object Oriented Developments (WOOD)*, pages 37–58. Elsevier, 2004.

[8] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming*, pages 428–452. Springer Berlin Heidelberg, 2005.

[9] Christopher Lyon Anderson. *Type Inference for Javascript*. PhD thesis, University of London, Imperial College London, Department of Computing, 2006.

[10] Esben Andreasen and Anders Møller. Determinacy in Static Analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 17–31. ACM, 2014.

[11] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile Resource Guarantees for Smart Devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 1–26. Springer, 2005.

[12] David Aspinall and Martin Hofmann. Another Type System for in-Place Update. In *11th European Symposium on Programming (ESOP)*, pages 36–52. Springer Berlin Heidelberg, 2002.

[13] David Aspinall, Patrick Maier, and Ian Stark. Monitoring external resources in Java MIDP. *Electronic Notes in Theoretical Computer Science*, 197(1):17–30, 2008.

[14] David Aspinall, Patrick Maier, and Ian Stark. Safety guarantees from explicit resource management. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, pages 52–71. Springer Berlin Heidelberg, 2008.

[15] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard – Fine-Grained Policy Enforcement for Untrusted Android Applications. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 213–231. Springer Berlin Heidelberg, 2014.

[16] Adam Barthe, Collin Jackson, and John C. Mitchell. Securing Frame Communication in Browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[17] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 49–54. ACM, 2011.

[18] Frédéric Besson, Guillaume Dufay, and Thomas Jensen. A formal model of access control for mobile interactive devices. In *Computer Security – ESORICS 2006*, pages 110–126. Springer Berlin Heidelberg, 2006.

[19] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 87–100. ACM, 2014.

[20] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *Proceedings of the 2011 Workshop on New Security Paradigms Workshop (NSPW*, pages 67–82. ACM, 2011.

[21] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 587–606. ACM, 2012.

[22] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *ACM Sigplan Notices*, pages 50–62. ACM, 2009.

[23] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. CRéPE: A System for Enforcing Fine-Grained Context-Related Policies on Android. *IEEE Transactions on Information Forensics and Security*, 7(5):1426 –1438, 2012.

[24] Progress Software Corporation. Telerik - Cordova Plugins. `http://plugins.telerik.com/cordova`, May 2016.

[25] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., Farnham, 2008.

[26] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35, 1998.

[27] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. AdSentry: Comprehensive and Flexible Confinement of JavaScript-based Advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pages 297–306. ACM, 2011.

[28] Spiridon Aristides Eliopoulos, Shriram Krishnamurthi, Joe Gibbs Politz, and Arjun Guha. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *Proceedings of the 20th USENIX conference on Security (SEC)*, pages 12–12. USENIX Association Berkeley, 2011.

[29] Karim O. Elish, Xiaokui Shu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Xuxian Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49:255–273, 2015.

[30] William Enck. Defending users against smartphone apps: Techniques and future directions. In *Information Systems Security*, pages 49–70. Springer Berlin Heidelberg, 2011.

[31] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, pages 235–245. ACM, 2009.

[32] Cameron Knight et al. GorillaScript. `http://ckknight.github.io/gorillascript`, April 2016.

[33] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999*, pages 32–45, 1999.

[34] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *CCS*. ACM, 2011.

[35] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to ask for permission. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, pages 7–7, 2012.

[36] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 75–86, 2011.

[37] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS)*, pages 3:1–3:14. ACM, 2012.

[38] Linux Foundation. Tizen project. `https://www.tizen.org/`, April 2014.

[39] Daniel Franzen and David Aspinall. Towards an amortized type system for JavaScript. In *SCSS*, pages 12–26. EPiC Series, vol. 30, 2014.

[40] Daniel Franzen and David Aspinall. PhoneWrap - Injecting the "How Often" into Mobile Apps. In *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS)*, pages 11–19. ceur-ws.org, 2016.

[41] Philippa Gardner, Gareth Smith, and Daiva Naudziuniene. JuS: Squeezing the Sense out of JavaScript Programs. *JSTools*, 2013.

[42] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 31–44. ACM, 2012.

[43] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[44] GitHub. Electron. `http://electron.atom.io`, May 2016.

[45] Google. AtScript. `https://docs.google.com/document/d/11YUzC-1d0V1-Q3V0fQ7KSit97HnZoKVygDxpWzEYW0U`, April 2016.

[46] Willem De Groef, Dominique Devriese, and Frank Piessens. Better Security and Privacy for Web Browsers: A Survey of Techniques, and a New Implementation. In *Formal Aspects of Security and Trust*, pages 21–38. Springer Berlin Heidelberg, 2012.

[47] Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, pages 151–168, 2009.

[48] Arjun Guha, Joe Gibbs Politz, and Shriram Krishnamurthi. Fluid object types. Technical report, Technical Report CS-11-04, Brown University, 2011.

[49] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *Proceedings of the 24th European conference on Object-oriented programming (ECOOP)*, pages 126–150. Springer Berlin Heidelberg, 2010.

[50] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State using Flow Analysis. In *Proceedings of the 20th European conference on Programming languages and systems (ESOP)*, pages 256–275. Springer Berlin Heidelberg, 2011.

[51] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *ACM SIGPLAN Notices*, pages 239–250. ACM, 2012.

[52] Phillip Heidegger and Peter Thiemann. Recency Types for Analyzing Scripting Languages. In *24th European Conference on Object-Oriented Programming (ECOOP)*, pages 200–224. Springer Berlin Heidelberg, 2010.

[53] David Herman and Cormac Flanagan. Status Report: Specifying JavaScript with ML. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 47–52. ACM, 2007.

[54] Martin Hofmann. A Type System for Bounded Space and Functional in-Place Update. In *9th European Symposium on Programming (ESOP)*, pages 258–289. Springer Berlin Heidelberg, 2000.

[55] Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 185–197. ACM, 2003.

[56] Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming (ESOP)*, pages 22–37. Springer Berlin Heidelberg, 2006.

[57] Martin Hofmann and Dulma Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In *Computer Science Logic*, pages 317–331. Springer Berlin Heidelberg, 2009.

[58] Martin Hofmann and Dulma Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In *Programming Languages and Systems*, pages 593–613. Springer Berlin Heidelberg, 2013.

[59] Adobe Systems Inc. PhoneGap. `http://phonegap.com/`, February 2014.

[60] Google Inc. Chrome Developers. `https://developer.chrome.com/apps`, April 2016.

[61] ECMA International. *ECMA-262: ECMAScript Language Specification 2015*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, sixth edition, 2015.

[62] Simon Holm Jensen. *Static Analysis for JavaScript*. PhD thesis, Aarhus University, Department of Computer Science, 2013.

[63] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the Eval that Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44. ACM, 2012.

[64] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Static Analysis (SAS)*, pages 238–255. Springer Berlin Heidelberg, 2009.

[65] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural Analysis with Lazy Propagation. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis (SAS)*, pages 320–339. Springer Berlin Heidelberg, 2010.

[66] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, page 3–14, 2012.

[67] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 66–77. ACM, 2014.

[68] Xing Jin, Lusha Wang, Tongbo Luo, and Wenliang Du. Fine-Grained Access Control for HTML5-Based Mobile Applications in Android. In *Information Security Conference (ISC)*, pages 309–318. Springer International Publishing, 2015.

[69] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 121–132. ACM, 2014.

[70] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: Designing a sound, configurable, and efficient static analyzer for JavaScript. *arXiv:1403.3996*, 2014.

[71] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type Refinement for Static Analysis of JavaScript. In *Proceedings of the 9th symposium on Dynamic languages (DLS)*, pages 17–26. ACM, 2013.

[72] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: Installing applications on an Android smartphone. In *Financial Cryptography and Data Security*, pages 68–79. Springer Berlin Heidelberg, 2012.

[73] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 194–206. ACM, 1973.

[74] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th symposium on Dynamic languages (DLS)*, pages 1–16. ACM, 2013.

[75] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, pages 2–16, 2005.

[76] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4):409–445, 1976.

[77] Francesco Logozzo and Herman Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation–Application to JavaScript Optimization. In *Compiler Construction (CC)*, pages 66–83. Springer, 2010.

[78] Mike Ter Louw, Phu H. Phung, Rohini Krishnamurti, and Venkat N. Venkatakrishnan. SafeScript: JavaScript Transformation for Policy Enforcement. In *Secure IT Systems, 18th Nordic Conference (NordSec)*, pages 67–83. Springer Berlin Heidelberg, 2013.

[79] Canonical Ltd. Ubuntu Phone. `http://www.ubuntu.com/phone/developers`, April 2016.

[80] Symphony Luo and Peter Yan. Fake apps: Feigning legitimacy. `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-fake-apps.pdf`, 2014. [last accessed Jan 2016].

[81] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 307–325. Springer Berlin Heidelberg, 2008.

[82] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *Information Security Technology for Applications, 15th Nordic Conference on Secure IT Systems (NordSec)*, pages 239–255. Springer Berlin Heidelberg, 2012.

[83] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval Begone!: Semi-Automated Removal of Eval from JavaScript Programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 607–620. ACM, 2012.

[84] L.A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *IEEE Symposium on Security and Privacy (SP)*, pages 481–496. IEEE, 2010.

[85] Microsoft. TypeScript. `https://www.typescriptlang.org/`, April 2016.

[86] Microsoft Corporation. Microsoft Security Intelligence Report, Volume 19. *`http://download.microsoft.com/download/4/4/C/44CDEF0E-7924-4787-A56A-16261691ACE3/Microsoft_Security_Intelligence_Report_Volume_19_English.pdf`*, 2015.

[87] John C. Mitchell. Coercion and Type Inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 175–185. ACM, 1984.

[88] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (Mobicom)*, pages 317–328. ACM, 2012.

[89] Mozilla. asm.js. `http://asmjs.org`, April 2016.

[90] Mozilla. FirefoxOS. `https://www.mozilla.org/en-GB/firefox/os/`, April 2016.

[91] Changhee Park, Hongki Lee, and Sukyoung Ryu. An Empirical Study on the Rewritability of the with Statement in JavaScript. In *2011 International Workshop on Foundations of Object-Oriented Languages (FOOL)*. ACM, 2011.

[92] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 346–356. ACM, 2015.

[93] K. Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *2011 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 720–729, 2011.

[94] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)*, pages 47–60. ACM, 2009.

[95] Michael Pradel and Koushik Sen. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP)*, pages 519–541, 2015.

[96] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 3–3. USENIX Association Berkeley, 2010.

[97] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *ACM Transactions on Web, Vol 1, No 3*, pages 11:1–11:32, 2007.

[98] John C. Reynolds. *Algol-like Languages*, chapter The Essence of Algol, pages 67–88. Birkhäuser Boston, 1997.

[99] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do. In *ECOOP 2011 – Object-Oriented Programming*, pages 52–78. Springer Berlin Heidelberg, 2011.

[100] Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. Flexible Access Control for JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 305–322, 2013.

[101] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 1–12. ACM, 2010.

[102] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP)*, page 224–238, 2012.

[103] Didier Rémy. Type inference for records in natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical aspects of object-oriented programming*, pages 67–95. MIT Press, 1994.

[104] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. *ACM SIGPLAN Notices*, pages 165–174, 2013.

[105] Charles Severance. JavaScript: Designing a Language in 10 Days. *Computer, IEEE Computer Society*, 45:7–8, 2012.

[106] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[107] Ian Stark. Reasons to believe: Digital evidence to guarantee trsustworthy mobile code. `www.homepages.inf.ed.ac.uk/stark/digital-evidence.pd`, 2009. [last accessed Jan 2016].

[108] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Renia Jeffers. Strong Mobility and Fine-Grained Resource Control NOMADS. In *Agent Systems, Mobile Agents, and Applications*, pages 2–15. Springer Berlin Heidelberg, 2000.

[109] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual Typing Embedded Securely in JavaScript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 425–437. ACM, 2014.

[110] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94 Parallel Architectures and Languages Europe*, number 817 in Lecture Notes in Computer Science, pages 398–413. Springer Berlin Heidelberg, 1994.

[111] Ankur Taly, Ulfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP)*, pages 363–378. IEEE, 2011.

[112] ACADINE Technologies. ACADINE, H5OS release. `https://www.acadine.com/product.html`, April 2016.

[113] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security Symposium*, pages 371–388, 2010.

[114] Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *14th European Symposium on Programming (ESOP)*, pages 408–422. Springer Berlin Heidelberg, 2005.

[115] Peter Thiemann and Phillip Heidegger. Recency Types for Dynamically-Typed, Object-Based Languages. In *16th International Workshop on Foundations of Object-Oriented Languages (FOOL)*. ACM, 2009.

[116] Philip Wadler. Linear Types Can Change the World! In *Programming Concepts and Methods*, pages 347–359. North, 1990.

[117] Shiyi Wei and Barbara G. Ryder. Practical Blended Taint Analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 336–346. ACM, 2013.

[118] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 27–27. USENIX Association, 2012.

[119] Tian Zhao. Type Inference for Scripting languages with Implicit Extension. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages (FOOL)*. ACM, 2010.

[120] Tian Zhao. Polymorphic Type Inference for Scripting Languages with Object Extensions. In *Proceedings of the 7th symposium on Dynamic languages (DLS)*, pages 37–50. ACM, 2011.